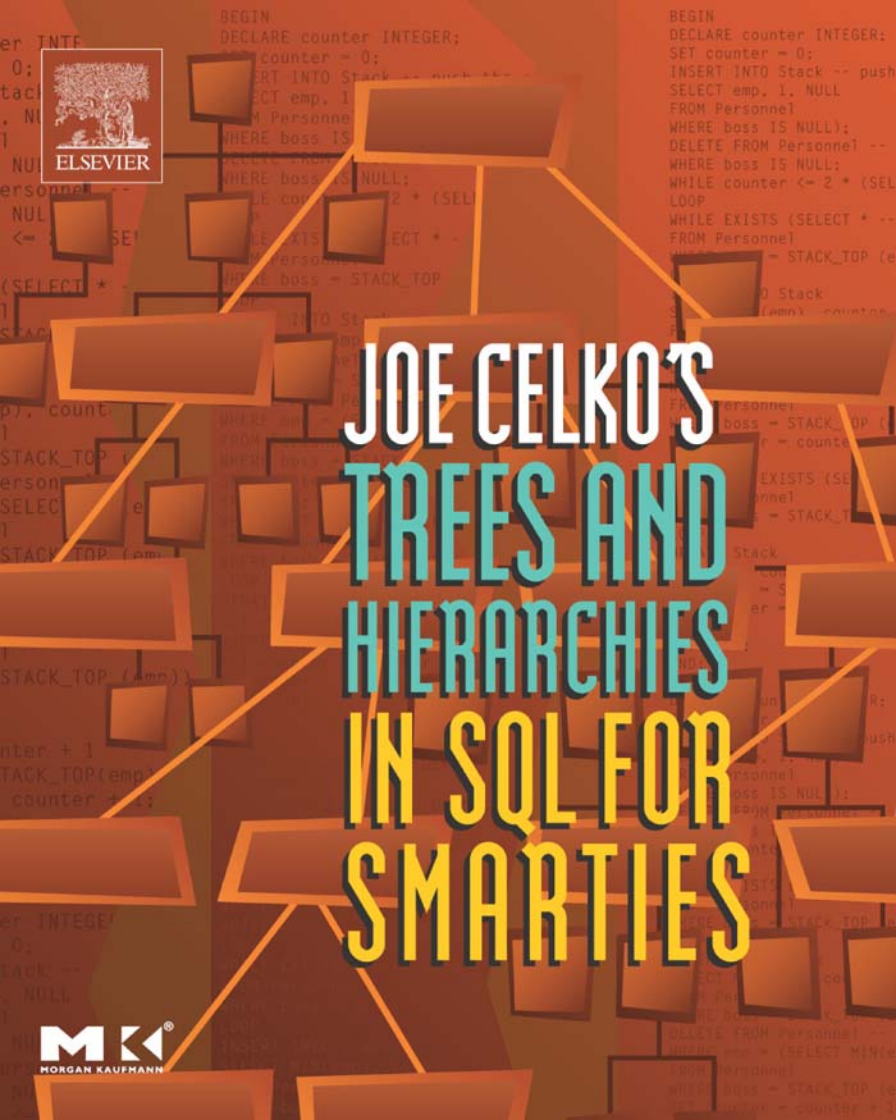
VISIT…


ELSEVIER
JOE CELKO'S
TREES AND
HIERARCHIES
IN SQL FOR
SMARTIES
MORGAN KAUFMANN

JOE CELKO'S TREES AND HIERARCHIES IN SQL FOR SMARTIES

This page intentionally left blank

JOE CELKO'S TREES AND HIERARCHIES IN SQL FOR SMARTIES

Joe Celko

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
MORGAN KAUFMANN PUBLISHERS IS AN IMPRINT OF ELSEVIER

MORGAN KAUFMANN PUBLISHERS

Acquisitions Editor	Lothlórien Homet
Publishing Services Manager	Andre Cuello
Project Manager	Anne B. McGee
Editorial Coordinator	Corina Derman
Cover Design	Side by Side Studios
Cover Image	Side by Side Studios
Composition	Kolam, Inc.
Copyeditor	Kolam USA
Proofreader	Kolam USA
Indexer	Kolam USA
Interior printer	The Maple-Vail Book Manufacturing Group
Cover printer	Phoenix Color Corp.

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

© 2004 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

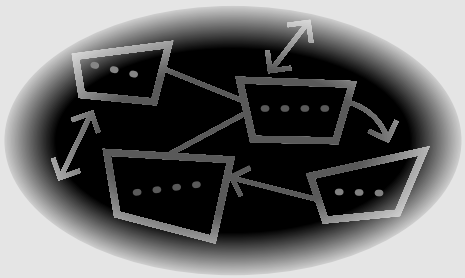
Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: *permissions@elsevier.com.uk*. You may also complete your request on-line via the Elsevier homepage (*http://elsevier.com*) by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data

Celko, Joe.
Joe Celko's Trees and hierarchies in SQL for smarties / Joe Celko.
p. cm.
Includes bibliographical references.
ISBN 1-55860-920-2
1. SQL (Computer program language) 2. Trees (Graph theory) I. Title: Trees and hierarchies in SQL for smarties. II. Title.
QA76.73.S67C435 2004
05.13'3—dc20 2004006193

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com*.

Printed in the United States of America
04 05 06 07 08 5 4 3 2 1



For Hilary and Kara
I love and believe in you both

This page intentionally left blank

CONTENTS

Introduction 1

1 Graphs, Trees, and Hierarchies 3

1.1 Modeling a Graph in a Program 4

1.1.1 Adjacency Lists for Graphs 6

1.1.2 Adjacency Arrays for Graphs 6

1.1.3 Finding a Path in General Graphs in SQL 7

1.2 Defining Trees and Hierarchies 11

1.2.1 Trees 11

1.2.2 Properties of Hierarchies 11

1.2.3 Types of Hierarchies 12

1.3 Note on Recursion 13

2 Adjacency List Model 17

2.1 The Simple Adjacency List Model 17

2.2 The Simple Adjacency List Model Is Not Normalized 19

2.2.1 UPDATE Anomalies 19

2.2.2 INSERT Anomalies 20

2.2.3 DELETE Anomalies 21

2.2.4 Structural Anomalies 22

2.3 Fixing the Adjacency List Model 22

2.3.1 Concerning the Use of NULLs 25

2.4 Navigation in Adjacency List Model 25

2.4.1 Cursors and Procedural Code 25

2.4.2 Self-joins 26

2.5 Inserting Nodes in the Adjacency List Model 28

2.6 Deleting Nodes in the Adjacency List Model 28

2.6.1 Deleting an Entire Subtree 28

2.6.2 Promoting a Subordinate after Deletion 30

2.6.3 Promoting an Entire Subtree after Deletion 30

2.7 Leveled Adjacency List Model 31

2.7.1 Numbering the Levels 32

2.7.2 Aggregation in the Hierarchy 33

3 Path Enumeration Models 35

3.1 Finding the Depth of the Tree 37
3.2 Searching for Subordinates 37
3.3 Searching for Superiors 38
3.4 Deleting a Subtree 39
3.5 Deleting a Single Node 39
3.6 Inserting a New Node 40
3.7 Splitting up a Path String 40
3.8 The Edge Enumeration Model 42
3.9 XPath and XML 43

4 Nested Set Model of Hierarchies 45

4.1 Finding Root and Leaf Nodes 48
4.2 Finding Subtrees 49
4.3 Finding Levels and Paths in a Tree 50
4.3.1 Finding the Height of a Tree 50
4.3.2 Finding Levels of Subordinates 50
4.3.3 Finding Oldest and Youngest Subordinates 56
4.3.4 Finding a Path 58
4.3.5 Finding Relative Position 58
4.4 Functions in the Nested Sets Model 59
4.5 Deleting Nodes and Subtrees 60
4.5.1 Deleting Subtrees 61
4.5.2 Deleting a Single Node 63
4.5.3 Pruning a Set of Nodes from a Tree 65
4.6 Closing Gaps in the Tree 66
4.7 Summary Functions on Trees 69
4.7.1 Iterative Parts Update 70
4.7.2 Recursive Parts Update 74
4.8 Inserting and Updating Trees 77
4.8.1 Moving a Subtree within a Tree 80
4.8.2 MoveSubtree, Second Version 84
4.8.3 Subtree Duplication 85
4.8.4 Swapping Siblings 88

4.9 Converting Nested Sets Model to Adjacency List 89
4.10 Converting Adjacency List to Nested Sets Model 90
4.11 Separation of Edges and Nodes 92
4.11.1 Multiple Structures 92
4.11.2 Multiple Nodes 93
4.12 Comparing Nodes and Structure 94
4.13 Nested Sets Code in Other Languages 98

5 Frequent Insertion Trees 101
5.1 The Datatype of (lft, rgt) 103
5.1.1 Exploiting the Full Range of Integers 103
5.1.2 FLOAT, REAL, or DOUBLE PRECISION Numbers 103
5.1.3 NUMERIC(p,s) or DECIMAL(p,s) Numbers 104
5.2 Computing the Spread to Use 104
5.2.1 Varying the Spread 107
5.2.2 Divisor Parameter 108
5.2.3 Divisor via Formula 108
5.2.4 Divisor via Table Lookup 109
5.2.5 Partial Reorganization 109
5.2.6 Rightward Spread Growth 111
5.3 Total Reorganization 113
5.3.1 Reorganization with Lookup Table 113
5.3.2 Reorganization with Recursion 117
5.4 Rational Numbers and Nested Intervals Model 119
5.4.1 Partial Order Mappings 120
5.4.2 Summation of Coordinates 123
5.4.3 Finding Parent Encoding and Sibling Number 126
5.4.4 Calculating the Enumerated Path and Distance between Nodes 128
5.4.5 Building a Hierarchy 132
5.4.6 Depth-first Enumeration by Left Interval Boundary 133
5.4.7 Depth-first Enumeration by Right Interval Boundary 134
5.4.8 All Descendants of a Node 134

6 The Linear Version of the Nested Sets Model 137
6.1 Insertion and Deletion 138
6.2 Finding Paths 140

6.3 Finding Levels 140

6.4 Summary 141

7 Binary Trees 143

7.1 Binary Tree Traversals 145

7.2 Binary Tree Queries 147

7.2.1 Find Parent of a Node 148

7.2.2 Find Subtree at a Node 149

7.3 Deletion from a Binary Tree 150

7.4 Insertion into a Binary Tree 150

7.5 Heaps 150

7.6 Binary Tree Representation of Multiway Trees 154

7.7 The Stern-Brocot Numbers 155

8 Other Models for Trees 157

8.1 Adjacency List with Self-references 157

8.2 Subordinate Adjacency List 158

8.3 Hybrid Models 159

8.3.1 Adjacency and Nested Sets Model 159

8.3.2 Nested Set with Depth Model 160

8.3.3 Adjacency and Depth Model 160

8.3.4 Computed Hybrid Models 161

8.4 General Graphs 164

8.4.1 Detecting Paths in a Convergent Graph 164

8.4.2 Detecting Directed Cycles 167

9 Proprietary Extensions for Trees 169

9.1 Oracle Tree Extensions 169

9.2 XDB Tree Extension 171

9.3 DB2 and the WITH Operator 172

9.4 Date's EXPLODE Operator 173

9.5 Tillquist and Kuo's Proposals 173

9.6 Microsoft Extensions 174

9.7 Other Methods 174

10 Hierarchies in Data Modeling 175

10.1 Types of Hierarchies 179

10.2 DDL Constraints 180

10.2.1 Uniqueness Constraints 180
10.2.2 Disjoint Hierarchies 183
10.2.3 Representing 1:1, 1:m, and n:m Relationships 186

11 Hierarchical Encoding Schemes 191

11.1 ZIP codes 191

11.2 Dewey Decimal Classification 192

11.3 Strength and Weaknesses 193

11.4 Shop Categories 194

11.5 Statistical Tools for Decision Trees 197

12 Hierarchical Database Systems (IMS) 199

12.1 Types of Databases 199

12.2 Database History 200

12.2.1 DL/I 202
12.2.2 Control Blocks 202
12.2.3 Data Communications 202
12.2.4 Application Programs 203
12.2.5 Hierarchical Databases 203
12.2.6 Strengths and Weaknesses 203

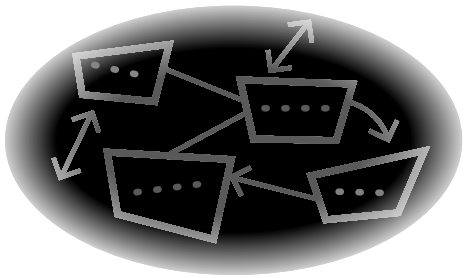
12.3 Sample Hierarchical Database 204

12.3.1 Departmental Database 206
12.3.2 Student Database 206
12.3.3 Design Considerations 206
12.3.4 Example Database Expanded 206
12.3.5 Data Relationships 208
12.3.6 Hierarchical Sequence 209
12.3.7 Hierarchical Data Paths 209
12.3.8 Database Records 210
12.3.9 Segment Format 211
12.3.10 Segment Definitions 212

12.4 Summary 213

Appendix: Readings and Resources **215**

Index **217**



Introduction

AN INTRODUCTION SHOULD give a noble purpose for writing a book. I should say that the purpose of this book is to help real programmers who have real problems in the real world. But the *real* reason this short book is being published is to save me the trouble of writing any more emails and posting more code on Internet Newsgroups. This topic has been hot on all the SQL-related websites and the solutions actually being used by most working programmers have been pretty bad. So why not collect everything I can find and put it in one place for the world to see?

In my book *SQL For Smarties* 2nd edition (Morgan-Kaufmann, 2000), I wrote a chapter on a programming technique for representing trees and hierarchies in SQL as nested sets. This technique has become popular enough that I have spent almost every month since *SQL For Smarties* was released explaining the technique in Newsgroups and personal emails. And people who have used it have been sending me emails with their programming tricks. Oh, I will still have a short chapter or two on trees in any future edition of *SQL for Smarties*, but this topic is worth this short monograph.

The first section of the book is a bit like an introductory college textbook on graph theory, so you might want to skip over it, if you are current on the subject. If you are not, then the theory there will explain some of the constraints that appear in the SQL code later. The middle sections deal with programming techniques and the end sections deal with related topics in computer programming.

The code in the book was checked using a SQL - 92 and SQL - 99 syntax validator program at the Mimer website (http://developer.mimer.com/validator/index.htm). I have used as much core SQL - 92 code as possible. When I needed procedural code in an example, I used SQL/PSM but tried to stay within a subset that can be easily translated into a vendor dialect (see Jim Melton's book, *Understanding SQL's Stored Procedures*, for details of this language [Morgan-Kaufmann, ISBN 0-55860-461-8, 1998]).

There are two major examples (and some minor ones) in this book. One is an organizational chart for an unnamed organization and the other is a parts explosion for a Frammis. Before anyone asks what a Frammis is, let me tell you that it is what holds all those Widgets that the MBA students were manufacturing in the fictional companies in their textbooks.

I invite corrections, additions, general thoughts, and new coding tricks at my email address (joe.celko@northface.edu) or my publisher's snail mail address.

Graphs, Trees, and Hierarchies

LET'S START WITH a little mathematical background. Graph theory is a branch of mathematics that deals with abstract structures, known as graphs. These are not the presentation charts that you get out of a spreadsheet package.

Very loosely speaking, a graph is a diagram of "dots" (called nodes or vertices) and "lines" (edges) that model some kind of "flow" or relationship. The edges can be undirected or directed. Graphs are *very* general models. In circuit diagrams the edges are the wires and the nodes are the components. On a road map the nodes are the towns and the edges are the roads. Flowcharts, organizational charts, and a hundred other common abstract models you see every day are all shown as graphs.

A directed graph allows a "flow" along the edges in one direction only, as shown by the arrowheads, whereas an undirected graph allows the flow to travel in both directions. Exactly what is flowing depends on what you are modeling with the graph.

The convention is that an edge must join two (and only two) nodes. This lets us show an edge as an ordered pair of nodes, such as (Atlanta, Boston) if we are dealing with a map, or (a, b) in a more abstract notation. There is an implication in a directed graph that the direction is shown by the ordering. In an undirected graph we know that (a, b) = (b, a), however.

A node can sit alone or have any number of edges associated with it. A node can also be self-referencing, as in (a, a).

The terminology used in graph theory will vary, depending on which book you had in your finite math class. The following list, in informal language, includes the terms I will use in this book.

Order of a graph: number of nodes in the graph

Degree: the number of edges at a node, without regard to whether the graph is directed or undirected

Indegree: the number of edges coming into a node in a directed graph

Outdegree: the number of edges leaving a node in a directed graph

Subgraph: a graph that is a subset of another graph's edges and nodes

Walk: a subgraph of alternating edges and nodes that are connected to each other in such a way that you can trace around it without lifting your finger

Path: a subgraph that does not cross over itself—there is a starting node with degree one, an ending node with degree one, and all other nodes have degree two. It is a special case of a walk. It is a "connect-the-dots" puzzle.

Cycle: a subgraph that "makes a loop," so that all nodes have degree two. In a directed graph all the nodes of a cycle have outdegree one and indegree one (Figure 1.1).

Connected graph: a graph in which all pairs of nodes are connected by a path. Informally, the graph is all in one piece.

Forest: a collection of separate trees. Yes, I am defining this term before we finally get to discussing trees.

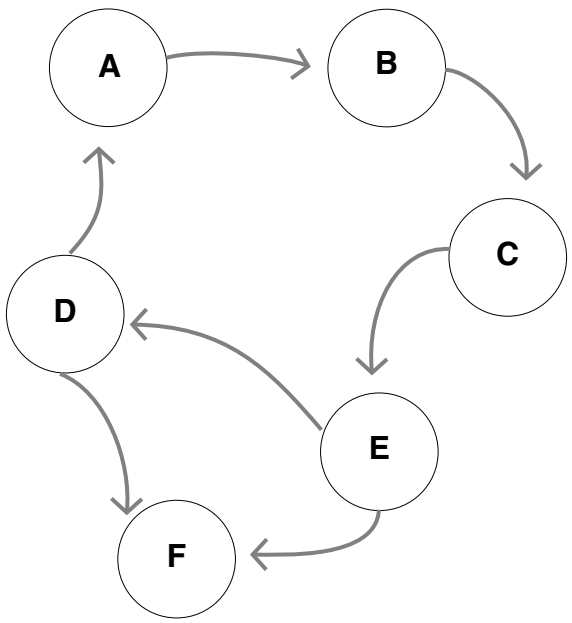
There are a lot more terms to describe special kinds of graphs, but frankly, we will not use them in this book. We are supposed to be learning SQL programming, not graph theory.

The strength of graphs as problem-solving tools is that the nodes and edges can be given extra attributes that adapt this general model to a particular problem. Edges can be assigned "weights," such as expected travel time for the roads on a highway map. Nodes can be assigned "colors" that put them into groups, such as men and women. Look around and you will see how they are used.

1.1 Modeling a Graph in a Program

Long before there was SQL, programmers represented graphs in the programming language that they had. People used pointer chains in assembly

Fig. 1.1



language or system development languages such as 'C' to build very direct representations of graphs with machine language instructions. However, unlike the low level system development languages, the later, higher-level languages, such as Pascal, LISP, and PL/I, did not expose the hardware to the programmer. Pointers in these higher level languages were abstracted to hide references to the actual physical storage and often required that the pointers point to variables or structures of a particular type. (See PL/I's ADDR() function, pointer datatypes, and based variables as examples of this kind of language construct.)

Traditional application development languages do not have pointers, but they often have arrays. In particular, FORTRAN only had arrays for a data structure; a good FORTRAN programmer could use them for just about anything. Early versions of FORTRAN did not have character-string data types—everything was either an integer or a floating-point number. This meant the model of a graph had to be created by numbering the nodes and using the node numbers as subscripts to index into the arrays.

Once the array techniques for graphs were developed, they became part of the "programmer's folklore" and were implemented in other languages. I will use a pseudocode to explain the techniques.

1.1.1 Adjacency Lists for Graphs

The adjacency list model represents the edges of the graph as pairs of nodes, similar to the following computer code:

```
DECLARE ARRAY GraphList
OF RECORD [edge CHAR(1), in_node INTEGER, out_node INTEGER];
```

With data:

```
GraphList
edge in_node out_node
'a'    1      2
'b'    2      3
'c'    4      2
'd'    1      4
```

The algorithms that we used were based on loops that made the connections between two edges, in which the in_node of one row was equal to the out_node of another row.

1.1.2 Adjacency Arrays for Graphs

Many of the computational languages had library functions for matrix operations; therefore it was logical to put the graph into an array where it could be manipulated with these functions.

Given (n) nodes, you could declare an (n) by (n) array with zeros and ones in the cells. A "one" meant that there was an edge between the two nodes represented by the row and column of the cell, and a "zero" meant that there was not.

You can actually represent a two-dimensional array; for example: A[0:5, 0:5], with a table like this:

```
CREATE TABLE Array_A
(edge CHAR(10) NOT NULL,
i INTEGER NOT NULL UNIQUE
  CHECK (i BETWEEN 0 AND 5),
j INTEGER NOT NULL UNIQUE
  CHECK (j BETWEEN 0 AND 5),
PRIMARY KEY (i, j));
```

I have a chapter in *SQL For Smarties* on how to do basic matrix math with such tables. However, because SQL was not meant to be used this way, the

code to implement the old Adjacency Array algorithms is rather baroque. Array was added to SQL-99 as a "collection type" for columns, but it is not widely implemented and has serious limitations—it is a vector, or one-dimensional array, and not a full multidimensional structure.

1.1.3 Finding a Path in General Graphs in SQL

There is a classic problem in graph theory that illustrates how expensive it can be to do general graphs in SQL. What we want is a list of paths from any two nodes in a directed graph in which the edges have a weight. The sum of these weights gives us the cost of each path so that we can pick the cheapest path.

The best way is probably to use the Floyd-Warshall or Johnson algorithm in a procedural language and load a table with the results. However, I want to do this in pure SQL as an exercise. Let's start with a simple graph and represent it as an adjacency list with weights on the edges.

```
CREATE TABLE Graph
(source CHAR(2) NOT NULL,
destination CHAR(2) NOT NULL,
cost INTEGER NOT NULL,
PRIMARY KEY (source, destination));
```

I obtained data for this table from the book *Introduction to Algorithms* by Cormen, Leiserson, and Rivest (Cambridge, Mass., MIT Press, 1990, p. 518; ISBN 0-262-03141-8). This book is very popular in college courses in the United States. I made one decision that will be important later—I added self-traversal edges—the node is both the source and the destination so the cost of those paths is zero.

```
INSERT INTO Graph
VALUES ('s', 's', 0),
       ('s', 'u', 3),
       ('s', 'x', 5),
       ('u', 'u', 0),
       ('u', 'v', 6),
       ('u', 'x', 2),
       ('v', 'v', 0),
       ('v', 'y', 2),
       ('x', 'u', 1),
       ('x', 'v', 4),
       ('x', 'x', 0),
```

```
('x', 'y', 6),
('y', 's', 3),
('y', 'v', 7),
('y', 'y', 0);
```

I am not happy about this approach, because I have to decide the maximum number of edges in a path before I start looking for an answer. However, this will work, and I know that a path will have no more than the total number of nodes in the graph. Let's create a table to hold the paths:

```
CREATE TABLE Paths
(step_1 CHAR(2) NOT NULL,
step_2 CHAR(2) NOT NULL,
step_3 CHAR(2) NOT NULL,
step_4 CHAR(2) NOT NULL,
step_5 CHAR(2) NOT NULL,
total_cost INTEGER NOT NULL,
path_length INTEGER NOT NULL,
PRIMARY KEY (step_1, step_2, step_3, step_4, step_5));
```

The step_1 node is where I begin the path. The other columns are the second step, third step, fourth step, and so forth. The last step column is the end of the journey. The total_cost column is the total cost, based on the sum of the weights of the edges, on this path. The path_length column is harder to explain, but for now let's just say that it is a count of the nodes visited in the path.

To keep things easier let's look at all the paths from "s" to "y" in the graph. The INSERT INTO statement for constructing that set looks like this:

```
INSERT INTO Paths
SELECT G1.source, it is 's' in this example
    G2.source,
    G3.source,
    G4.source,
    G4.destination, it is 'y' in this example
    (G1.cost + G2.cost + G3.cost + G4.cost),
    (CASE WHEN G1.source
        NOT IN (G2.source, G3.source, G4.source)
      THEN 1 ELSE 0 END
    + CASE WHEN G2.source
        NOT IN (G1.source, G3.source, G4.source)
```

```
       THEN 1 ELSE 0 END
    + CASE WHEN G3.source
        NOT IN (G1.source, G2.source, G4.source)
       THEN 1 ELSE 0 END
    + CASE WHEN G4.source
        NOT IN (G1.source, G2.source, G3.source)
       THEN 1 ELSE 0 END)
FROM Graph AS G1, Graph AS G2, Graph AS G3, Graph AS G4
WHERE G1.source = 's'
  AND G1.destination = G2.source
  AND G2.destination = G3.source
  AND G3.destination = G4.source
  AND G4.destination = 'y';
```

I put in "s" and "y" as the source and destination of the path and made sure that the destination of one step in the path was the source of the next step in the path. This is a combinatorial explosion, but it is easy to read and understand.

The sum of the weights is the cost of the path, which is easy to understand. The path_length calculation is a bit harder. This sum of CASE expressions looks at each node in the path. If it is unique within the row, it is assigned a value of one; if it is not unique within the row, it is assigned a value of zero.

All paths will have five steps in them, because that is the way the table is declared. However, what if a path shorter than five steps exists between the two nodes? That is where the self-traversal rows are used! Consecutive pairs of steps in the same row can be repetitions of the same node.

Here is what the rows of the paths table look like after this INSERT INTO statement, ordered by descending path_length, and then by ascending cost:

Paths

step_1	step_2	step_3	step_4	step_5	total_cost	path_length
s	s	x	x	y	11	0
s	s	s	x	y	11	1
s	x	x	x	y	11	1
s	x	u	x	y	14	2
s	s	u	v	y	11	2
s	s	u	x	y	11	2
s	s	x	v	y	11	2
s	s	x	y	y	11	2
s	u	u	v	y	11	2

(cont.)

step_1	step_2	step_3	step_4	step_5	total_cost	path_length
s	u	u	x	y	11	2
s	u	v	v	y	11	2
s	u	x	x	y	11	2
s	x	v	v	y	11	2
s	x	x	v	y	11	2
s	x	x	y	y	11	2
s	x	y	y	y	11	2
s	x	y	v	y	20	4
s	x	u	v	y	14	4
s	u	v	y	y	11	4
s	u	x	v	y	11	4
s	u	x	y	y	11	4
s	x	v	y	y	11	4

Many of these rows are equivalent to each other. For example, the paths ('s', 'x', 'v', 'v', 'y', 11, 2) and ('s', 'x', 'x', 'v', 'y', 11, 2) are both really the same path as ('s', 'x', 'v', 'y').

In this example the total_cost column defines the cost of a path, so we can eliminate some of the paths from the table with this statement, if we want the lowest cost.

```
DELETE FROM Paths
WHERE total_cost
 > (SELECT MIN(total_cost)
    FROM Paths);
```

In this example, it got rid of three out of 22 possible paths. Let's consider another cost factor: the number of paths. People do not like to change airplanes or trains en route to their destination. If they can go from Amsterdam to New York without changing planes, for the same cost, they are happy. This is where that path_length column comes in. It is a quick way to remove the paths that have more edges than they need to get the job done.

```
DELETE FROM Paths
WHERE path_length
    > (SELECT MIN(path_length)FROM Paths);
```

In this case that last DELETE FROM statement will reduce the table to one row ('s', 's', 'x', 'x', 'y', 11, 0), which reduces to ('s', 'x', 'y'). This single remaining row is very convenient for my demonstration, but if you look at the table, you will see that there was also a subset of equivalent rows that had higher path_length numbers.

```
('s', 's', 's', 'x', 'y', 11, 1)
('s', 'x', 'x', 'x', 'y', 11, 1)
('s', 'x', 'x', 'y', 'y', 11, 2)
('s', 'x', 'y', 'y', 'y', 11, 2)
```

Your task is to write code to handle equivalent rows. Hint: the duplicate nodes will always be contiguous across the row.

1.2 Defining Trees and Hierarchies

There is an important difference between a tree and a hierarchy, which has to do with inheritance and subordination. Trees are a special case of graphs; hierarchies are a special case of trees. Let's start by defining trees.

1.2.1 Trees

Trees are graphs that have the following properties:

1. A tree is a connected graph that has no cycles. A connected graph is one in which there is a path between any two nodes. No node sits by itself, disconnected from the rest of the graph.
2. Every node is the root of a subtree. The most trivial case is a subtree of only one node.
3. Every two nodes in the tree are connected on one (and only one) path.
4. A tree is a connected graph that has one less edge than it has nodes.

In a tree, when an edge (a, b) is deleted, the result is a forest of two disjointed trees. One tree contains node (a) and the other contains node (b).

There are other properties, but this list gives us enough information for writing constraints in SQL. Remember, this is a book about programming, not graph theory. Therefore you will get just enough to help you write code, but not enough to be a mathematician.

1.2.2 Properties of Hierarchies

A hierarchy is a directed tree with extra properties: subordination and inheritance.

A hierarchy is a common way to organize a great many things, but the examples in this book will be organizational charts and parts explosions. These are two common business applications and can be easily understood by anyone without any special subject area knowledge. In addition, they demonstrate that

the relationship represented by the edges of the graph can run from the root or up to the root.

In an organizational chart authority starts at the root, with the president of the enterprise, head of the army, or whatever the organization is, and it flows downward. Look at a military chain of command. If you are a private and your sergeant is killed, you still have to take orders from your captain. Subordination is inherited from the root downward.

In a parts explosion the relationship we are modeling runs "up the tree" to the root, or final assembly. If you are missing any subassembly, you cannot get a final assembly.

Inheritance, either to or from the root, is the most important property of a hierarchy. This property does not exist in an ordinary tree. If I delete an edge in a tree, I now have two separate trees.

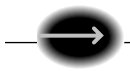
Another property of a hierarchy is that the same node can play many different roles. In an organizational chart one person might hold several different jobs; in a parts explosion the same kind of screw, nut, or washer will appear in many different subassemblies. Moreover, the same subassembly can appear in many places. To make this more concrete, imagine a restaurant with a menu. The menu disassembles into dishes, and each dish disassembles into ingredients, and each ingredient is either simple (e.g., salt, pepper, flour), or it is a recipe, itself, such as béarnaise sauce or hollandaise sauce. These recipes might include further recipes. For example, béarnaise sauce is essentially hollandaise, with vinegar substituted for the water, and the addition of shallots, tarragon, chervil, and (sometimes) parsley, thyme, bay leaf, and cayenne pepper.

Hierarchies have roles that are filled by entities. This role property does not exist in a tree; each node appears once in a tree and is unique.

1.2.3 Types of Hierarchies

Getting away from looking at the world from the viewpoint of a casual mathematician, let's look at it from the viewpoint of a casual database systems designer. What kinds of data situations will I want to model? Looking at the world from a very high level, I can see following four kinds of modeling problems:

1. Static nodes and static edges. For example, a chart of accounts in an accounting system will probably not change much over time. This is probably best done with a hierarchical encoding scheme rather than a table. We will talk about such encoding schemes later in this book.



2. Static nodes and dynamic edges, For example, an Internet Newsgroup message board. Obviously you cannot add a node to a tree without adding an edge, but the content of the messages (nodes) never change once they are posted; however, new replies can be posted as subordinates to any existing message (edge).
3. Dynamic nodes and static edges. This is the classic organizational chart in which the organization stays the same, but the people holding the offices rotate frequently. This is assuming that your company does not reorganize more often than its personnel turns over.
4. Dynamic nodes and dynamic edges. Imagine that you have a graph model of a communications or transportation network. The traffic on the network is constantly changing. You want to find a minimal spanning tree based on the current traffic and update that tree as the nodes and edges come on and off the network. To make this a little less abstract, the fastest path from the fire station to a particular home address will not necessarily be the same route at 05:00 Hrs as it will be at 17:00 Hrs. Once the fire is put out, the node that represented the burning house can disappear from the tree and the next fire location becomes a to which we must find a path.

Looking at the world from another viewpoint, we might classify hierarchies by usage—as either searching or reporting. An example of a searching hierarchy is the Dewey Decimal system in a library. You move from the general classifications to a particular book—down the hierarchy. An example of a reporting hierarchy is an accounting system. You move from particular transactions to summaries by general categories (e.g., assets, liabilities, equity)—up the hierarchy.

You might pick a different tree model for a table in each of these situations to get better performance. It can be a very hard call to make, and it is hard to give even general advice. Hopefully, I can show you the tradeoffs and you can make an informed decision.

1.3 Note on Recursion

I am going to take a little time to explain recursion, because trees are a recursive data structure and can be accessed by recursive algorithms. Many commercial programmers who are old enough to spell "Cobol" in uppercase letters are not familiar with the concept of recursion. Recursion does not appear in early programming languages. Even when it did, or was added later,

as was the case in IBM's MVS Cobol product in 1999, most programmers do not use it.

There is an old geek joke that gives the dictionary definition:

> Recursion = (REE - kur - shun) self-referencing; also see recursion.

This is pretty accurate, if not all that funny. A recursive structure is composed of smaller structures of the same kind. Thus a tree is composed of subtrees. You finally arrive at the smallest possible subtrees, the leaf nodes—a subtree of size one.

A recursive function is also like that; part of its work is done by invoking itself until it arrives at the smallest unit of work for which it can return an answer. Once it gets the lowest level answer, it passes it back to the copy of the function that called it, so that copy can finish its computations, and so forth until we have gotten back up the chain to the first invocation that started it all. It is very important to have a halting condition in a recursive function for obvious reasons.

Perhaps the idea will be easier to see with a simple example. Let's reverse a string with the following function:

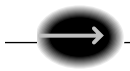
```
CREATE FUNCTION Reverse (IN instring VARCHAR(20))
RETURNS VARCHAR(20)
LANGUAGE SQL
DETERMINISTIC
BEGIN -- recursive function
IF CHAR_LENGTH(instring) IN (0, 1) -- halt condition
THEN RETURN (instring);
ELSE RETURN -- flip the two halves around, recursively
   (Reverse(SUBSTRING (instring FROM (CHAR_LENGTH(instring)/2 + 1))
   || Reverse(SUBSTRING (instring FROM 1 FOR
CHAR_LENGTH(instring)/2))));
END IF;
END;
```

Given the string 'abcde', the first call becomes:

```
Reverse('de') || Reverse('abc')
```

This becomes:

```
(Reverse(Reverse('e') || Reverse('d'))
|| Reverse(Reverse('c') || Reverse('ab'))
```



This becomes:

```
((‘e’||’d’)
|| ((‘c’) || Reverse((Reverse(‘b’) || Reverse(‘a’))))
```

This becomes:

```
((‘e’||‘d’) || (‘c’ || (‘b’||‘a’)))
```

This finally becomes:

```
‘edcba’
```

In the case of trees we will test to see if a node is either the root or a leaf node as our halting conditions. The rest of the time we are dealing with a subtree, which is just another tree. This is why a tree is called a “recursive structure.”

This page intentionally left blank

Adjacency List Model

IN THE EARLY days of System R at IBM one of the arguments against a relational database was that SQL could not handle hierarchies the way IMS could (see Chapter 12), and would therefore not be practical for large databases. It might have a future as an ad hoc query language, but that was the best that could be expected of it.

In a short paper Dr. E. F. Codd described a method for showing hierarchies in SQL that consisted of a column for the boss and another column for the employee in the relationship. It was a direct implementation in a table of the Adjacency List Model of a graph. Oracle was the first commercial database to use SQL, and the sample database that comes with their product, nicknamed the "Scott/Tiger" database in the trade because of its default user and password codes, uses an adjacency list model in a combination Personnel/Organizational chart table. The organizational structure and the personnel data are mixed together in the same row.

This model stuck for several reasons, other than just Dr. Codd's and Oracle's seeming endorsements. It is probably the most natural way to convert from an IMS database or from a procedural language to SQL if you have been a procedural programmer all of your life.

2.1 The Simple Adjacency List Model

In Oracle's Scott/Tiger personnel table the "linking column" is the employee identification number of the immediate boss of each employee. The president

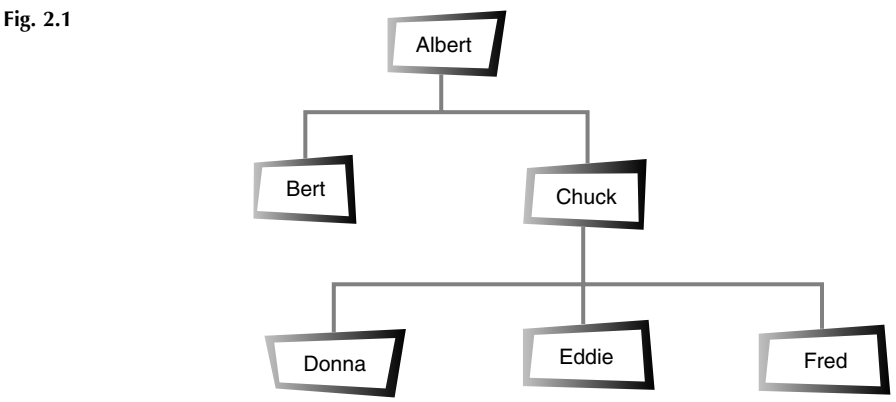


Fig. 2.1

of the company has a NULL for his boss. Here is an abbreviated version of such a Personnel/Organizational chart table (Figure 2.1):

```
CREATE TABLE Personnel_OrgChart
(emp VARCHAR(10) NOT NULL PRIMARY KEY,
boss VARCHAR(10), -- null means root
salary DECIMAL(6,2) NOT NULL,
... );
```

Personnel_OrgChart

emp	boss	salary
'Albert'	NULL	1000.00
'Bert'	'Albert'	900.00
'Chuck'	'Albert'	900.00
'Donna'	'Chuck'	800.00
'Eddie'	'Chuck'	700.00
'Fred'	'Chuck'	600.00

The use of a person's name for a key is not a good programming practice, but let's ignore that point for now; it will make the discussion easier. The table also needs a UNIQUE constraint to enforce the hierarchical relationships among the nodes. This is not a flaw in the adjacency list model per se, but this is how I have seen most programmers program the adjacency list model. In fairness, one reason for not having all of the needed constraints is that most SQL products did not have such features until their later versions. The constraints that should be used are complicated, and we will get to them after this history lesson.

I am first going to attack a "straw man," which shows up more than it should in actual SQL programming, and then I'm going make corrections to that initial adjacency list model schema. Finally, I want to show some actual flaws in the adjacency list model after it has been corrected.

2.2 The Simple Adjacency List Model Is Not Normalized

There is a horrible truth about the simple adjacency list model that nobody noticed. It is not a normalized schema. The short definition of normalization is that all data redundancy has been removed and it is safe from data anomalies. I coined the phrase that a normalized database has "one simple fact, in one place, one time" as a mnemonic for three characteristics we want in a data model. What we want is to bring this into Domain_key Normal Form (DKNF).

We will go into detail shortly, but for now consider that the typical adjacency list model table includes information about the node (the salary of the employee in this example), as well as who its boss (boss) is in each row. This means that you have a mixed table of entities (personnel) and relationships (organization), and thus its rows are not properly formed facts. So much for the first characteristic.

The second characteristic of a normalized table is that each fact appears "in one place" in the schema (i.e., it belongs in one row of one table), but the subtree of each node can be in more than one row. The third characteristic of a normalized table is that each fact appears "one time" in the schema (i.e., you want to avoid data redundancy). If both of these conditions are violated, we can have anomalies.

2.2.1 UPDATE Anomalies

Let's say that "Chuck" decides to change his name to "Charles," so we have to update the Personnel_OrgChart table:

```
UPDATE Personnel_OrgChart
  SET emp = 'Charles'
WHERE emp = 'Chuck';
```

However, that does not work. We want the table to look like this:

Personnel_OrgChart

emp	boss	salary
'Albert'	NULL	1000.00
'Bert'	'Albert'	900.00

(cont.)

emp	boss	salary	
'Charles'	'Albert'	900.00	<== change as employee
'Donna'	'Charles'	800.00	<== change as boss #1
'Eddie'	'Charles'	700.00	<== change as boss #2
'Fred'	'Charles'	600.00	<== change as boss #3

Four rows are affected by this UPDATE statement. If a Declarative Referential Integrity REFERENCES clause was used, then an ON UPDATE CASCADE clause with a self-reference could make the three "boss role" changes automatically. Otherwise, the programmer has to write two UPDATE statements.

```
BEGIN ATOMIC
UPDATE Personnel_OrgChart
   SET emp  = 'Charles'
 WHERE emp  = 'Chuck';
UPDATE Personnel_OrgChart
   SET boss = 'Charles'
 WHERE boss = 'Chuck';
END;
```

or, if you prefer, one UPDATE statement that hides the logic in a faster, but convoluted, CASE expression:

```
UPDATE Personnel_OrgChart
  SET emp = CASE WHEN emp = 'Chuck'
                 THEN 'Charles'
                 ELSE emp END,
      boss = CASE WHEN boss = 'Chuck'
                 THEN 'Charles'
                 ELSE boss END
WHERE 'Chuck' IN (boss, emp);
```

As you can see from these examples, this is not a simple change of just one fact.

2.2.2 INSERT Anomalies

The simple adjacency list model has no constraints to preserve subordination. Therefore you can easily corrupt the Personnel_OrgChart with a few simple insertions, as follows:

```
- make a cycle in the graph
INSERT INTO Personnel_OrgChart VALUES ('Albert', 'Fred', 100.00);
```

Obviously you can create cycles by inserting an edge between any two existing nodes.

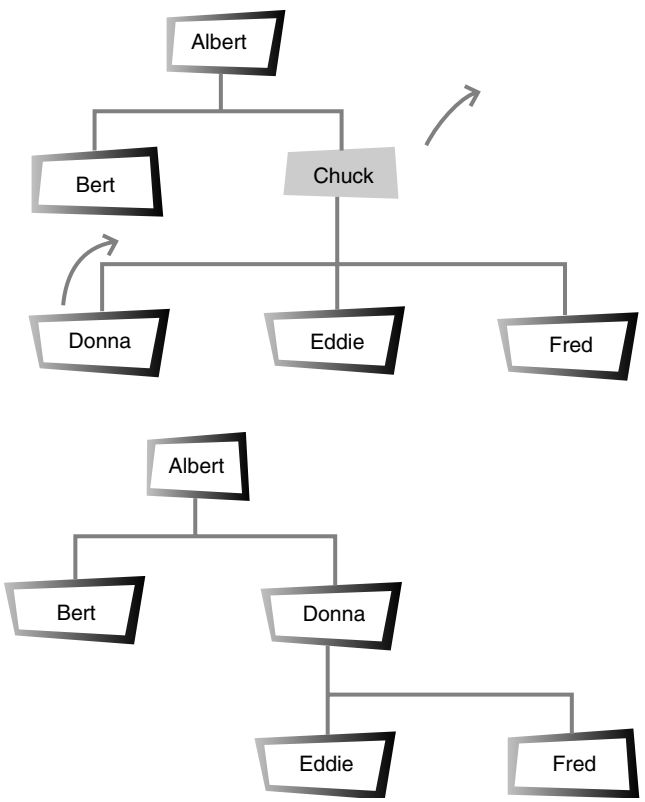
2.2.3 DELETE Anomalies

The simple adjacency list model does not support inheritance of subordination. Deleting a row will split the tree into several smaller trees, as for example:

```
DELETE FROM Personnel_OrgChart WHERE emp = 'Chuck';
```

Suddenly, 'Donna', 'Eddie', and 'Fred' find themselves disconnected from the organization and no longer reporting indirectly to 'Albert' (Figure 2.2). In fact, they are still reporting to 'Chuck', who does not exist anymore! Using an ON DELETE CASCADE referential action or a TRIGGER could cause the entire subtree to disappear—probably a bad surprise for Chuck's former subordinates.

Fig. 2.2



2.2.4 Structural Anomalies

Finally, we need to preserve the tree structure in the table. We need to be sure that there is only one NULL in the structure, but the simple adjacency list model does not protect against multiple NULLs or from cycles.

```
-- self-reference
INSERT INTO Personnel_OrgChart (boss, emp) VALUES (a, a);

-- simple cycle
INSERT INTO Personnel_OrgChart (boss, emp) VALUES (c, b);
INSERT INTO Personnel_OrgChart (boss, emp) VALUES (b, c);
```

The problem is that the adjacency list model is actually a general model for *any* graph. A tree is a special case of a graph, so you need to restrict the adjacency list model a bit to be sure that you do have only a tree. Otherwise, you will have Domain Key Normal Form (DKNF) problems

2.3 Fixing the Adjacency List Model

In fairness, I have been kicking a straw man. These flaws in the simple adjacency list model can be overcome with a redesign of the schema.

First, the personnel list and the organizational chart could, and should, be modeled as separate tables. The personnel table contains the facts about the people (entities) who we have as our personnel, and the organizational chart tells us how the job positions within the company are organized (relationships), regardless of whom—if anyone—holds what position. It is the difference between the office and the person who holds that office.

```
CREATE TABLE Personnel
(emp_nbr INTEGER DEFAULT 0 NOT NULL PRIMARY KEY,
emp_name VARCHAR(10) DEFAULT '{ {vacant} }' NOT NULL,
address VARCHAR(35) NOT NULL,
birth_date DATE NOT NULL,
...);
```

I am assuming that we have a dummy employee named '{ {vacant} }' with a dummy employee number of zero. It makes reports look nicer; however, you have to add more constraints to handle this missing value marker.

The information about the positions within the company goes into a second table, as follows:

```
CREATE TABLE OrgChart
(job_title VARCHAR(30) NOT NULL PRIMARY KEY,
emp_nbr INTEGER DEFAULT 0 -- zero is vacant position
    NOT NULL
    REFERENCES Personnel(emp_nbr)
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE,
boss_emp_nbr INTEGER -- null means root node
      REFERENCES Personnel(emp_nbr),
UNIQUE (emp_nbr, boss_emp_nbr),
salary DECIMAL (12,4) NOT NULL CHECK (salary >= 0.00),
...);
```

Notice that you still need constraints between, and within, the tables to enforce the tree properties and to make sure that a position is not held by someone who is not an employee of the company.

The most obvious constraint is to prohibit a single node cycle in the graph.

```
CHECK (boss_emp_nbr <> emp_nbr) - cannot be your own boss!
```

However, that does not work because of the dummy employee number of zero for vacant positions.

```
CHECK ((boss_emp_nbr <> emp_nbr) OR (boss_emp_nbr = 0 AND
emp_nbr = 0))
```

Longer cycles are prevented with a UNIQUE (emp, boss) constraint that limits an employee to one (and only one) boss.

We know that the number of edges in a tree is the number of nodes minus one, therefore this is a connected graph. That constraint looks like this:

```
CHECK ((SELECT COUNT(*) FROM OrgChart) - 1 -- edges
= (SELECT COUNT(boss_emp_nbr) FROM OrgChart)) -- nodes
```

The COUNT (boss_emp_nbr) will drop the NULL in the root row. That gives us the effect of having a constraint to check for one NULL:

```
CHECK((SELECT COUNT(*) FROM OrgChart WHERE boss_emp_nbr IS NULL)
= 1)
```

This is a necessary condition, but it is not a sufficient one. Consider this data, in which 'Donna' and 'Eddie' are in a cycle, and that cycle is not in the tree structure.

emp_name	boss_name
'Albert'	NULL
'Bert'	'Albert'
'Chuck'	'Albert'
'Donna'	'Eddie'
'Eddie'	'Donna'

One approach would be to remove all the leaf nodes and repeat this procedure until the tree is reduced to an empty set. If the tree does not reduce to an empty set, then there is a disconnected cycle.

```
CREATE FUNCTION TreeTest() RETURNS CHAR(6)
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
-- put a copy in a temporary table
INSERT INTO Tree SELECT emp, boss FROM Personnel_OrgChart;
-- prune the leaves
WHILE (SELECT COUNT(*) FROM Tree) - 1
    = (SELECT COUNT(boss) FROM Tree)
   DO DELETE FROM Tree
       WHERE Tree.emp
             NOT IN (SELECT T2.boss
                       FROM Tree AS T2
                      WHERE T2.boss IS NOT NULL);
       IF NOT EXISTS (SELECT * FROM Tree)
       THEN RETURN ('Tree');
       ELSE RETURN ('Cycles');
       END IF;
END WHILE;
END;
```

These constraints will need to be deferred in some situations; in particular, if we reorganize a position out of existence, we need to remove it from the organizational chart table and make a decision about its subordinates. We will deal with that problem in another section. The original Personnel_OrgChart is easy to reconstruct with following VIEW for reporting purposes:

```
CREATE VIEW Personnel_OrgChart (emp_nbr, emp, boss_emp_nbr, boss)
AS SELECT E1.emp_nbr, E1.emp_name, E1.boss_emp_nbr, B1.emp_name
     FROM Personnel AS E1, Personnel AS B1, OrgChart AS O1
```

```
 WHERE B1.emp_nbr = P1.boss_emp_nbr
   AND E1.emp_nbr = P1.emp_nbr;
```

2.3.1 Concerning the Use of NULLs

I have shown a NULL-able boss column in my examples, in which the NULL means that this row is the root of the tree; that is, it has no boss above it in the hierarchy. Although this is the most common representation, it is not the only way to model a tree. The alternatives include:

1. Use NULLs for the subordinates of leaf nodes. This leads to slightly different logic in many of the queries, reversing the "flow" of NULL checking.
2. Disallow NULLs altogether. This will record only the edges of the graph in the table. Again, the logic would change. The root would have to be detected by looking for the one node that is only a boss who reports to a dummy value of some kind and is never an employee, as follows:

```
SELECT DISTINCT boss_emp_nbr
  FROM OrgChart
 WHERE boss_emp_nbr NOT IN (SELECT emp_nbr FROM OrgChart);
```

In many ways I would prefer option 2 in the aforementioned list of alternatives; however, using the (NULL, <root>) convention guarantees that all employees show up in the emp_nbr column, which makes many queries much easier to write.

This convention was not done for that reason; historically, the boss was considered an attribute of the employee in the data model.

2.4 Navigation in Adjacency List Model

The fundamental problem with the adjacency list model is that it requires navigation. There is no general way to extract a complete subtree.

2.4.1 Cursors and Procedural Code

The practical problem is that, in spite of the existing SQL standards, every SQL product has slightly different proprietary cursor syntax. The general format is to follow the chain of (emp_nbr, boss_emp_nbr) values in a loop. This makes going down the tree fairly simple; however, aggregation of subtrees for reporting is very slow for large trees.

This approach is fairly simple if you start at the leaf nodes and travel to the root node of the tree structure.

```
CREATE PROCEDURE UpTreeTraversal (IN current_emp_nbr INTEGER)
LANGUAGE SQL
DETERMINISTIC
WHILE EXISTS
     (SELECT *
        FROM OrgChart AS T1
       WHERE current_emp_nbr = T1.emp_nbr)
  DO BEGIN
   -- take some action on the current node of the traversal
    CALL SomeProc (current_emp_nbr);
   -- go up the tree toward the root
    SET current_emp_nbr
        = (SELECT T1.boss_emp_nbr
             FROM OrgChart AS T1
            WHERE current_emp_nbr = T1.emp_nbr);
     END;
END WHILE;
```

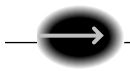
2.4.2 Self-joins

The other method of doing a tree traversal is to do multiple self-joins, with each copy of the tree representing a level in the Personnel_OrgChart.

```
SELECT O1.emp AS e1, O2.emp AS e2, O3.emp AS e3
FROM Personnel_OrgChart AS O1, Personnel_OrgChart AS O2,
     Personnel_OrgChart AS O3
WHERE O1.emp = O2.boss
  AND O2.emp = O3.boss
  AND O1.emp = 'Albert';
```

This code is limited to a known depth of traversal, which is not always possible. This sample query produces this result table. The paths shown are those that are exactly three levels deep.

e1	e2	e3
'Albert'	'Chuck'	'Donna'
'Albert'	'Chuck'	'Eddie'
'Albert'	'Chuck'	'Fred'



You can improve this query with the use of LEFT OUTER JOINs.

```
SELECT O1.emp AS e1, O2.emp AS e2, O3.emp AS e3, O4.emp AS e4
  FROM Personnel_OrgChart AS O1
       LEFT OUTER JOIN
       Personnel_OrgChart AS O2
       ON O1.emp = O2.boss
          LEFT OUTER JOIN
          Personnel_OrgChart AS O3
          ON O2.emp = O3.boss
           LEFT OUTER JOIN
          Personnel_OrgChart AS O4
          ON O3.emp = O4.boss
WHERE O1.emp = 'Albert';
```

Any paths at a particular level that are not in the table will be displayed as NULLs, so that this query can be put into a VIEW and invoked.
Notice what it produces:

e1	e2	e3	e4
'Albert'	'Bert'	NULL	NULL
'Albert'	'Chuck'	'Donna'	NULL
'Albert'	'Chuck'	'Eddie'	NULL
'Albert'	'Chuck'	'Fred'	NULL

This actually gives you all the subtree paths under 'Albert', to a fixed depth of three. The pattern can be extended, but performance will also go down. Most SQL products have a point at which the optimizer chokes on either the number of tables in a FROM clause or on the levels of self-reference in a query.

Aggregation based on self-joins is a nightmare. You have to build a table with one column that has the unique keys of the subtree and use it to find the rows to be used in the aggregate calculations. One way to "flatten" the table is to use an auxiliary table, called Sequence, which contains the single column sequence of integers from 1 to (n), where (n) is a sufficiently large number.

```
SELECT MAX(CASE
  WHEN seq = 1 THEN e1
  WHEN seq = 2 THEN e2
  WHEN seq = 3 THEN e3
  WHEN seq = 4 THEN e4
  ELSE NULL END)
```

```
  FROM (Sequence AS S1
     CROSS JOIN
     << Personnel_OrgChart query as above>>
     ) AS X (e1, e2, e3, e4)
WHERE seq BETWEEN 1 AND 4;
```

As you can see, this approach quickly becomes insanely convoluted and you do not gain generality.

2.5 Inserting Nodes in the Adjacency List Model

Insertion is the strong point of the adjacency list model. You just insert the (emp_nbr, boss_emp_nbr) pairs into the table and you are done; no other rows need to be changed.

2.6 Deleting Nodes in the Adjacency List Model

Removing a leaf node is easy—just remove the row from the tree structure table. All of the tree properties are preserved and no constraints will be violated.

The code for deleting nodes inside the tree is much more complex. First, you must make a decision about how to handle the surviving subordinates, by choosing from the following three basic approaches:

1. The Ancient Egyptian school of management: When a node is removed, all of the subordinates are removed. When a pharaoh dies, you bury all his slaves with him.
2. Send the orphans to Grandmother: The subordinates of the deleted node became immediate subordinates of their boss's boss.
3. The oldest son takes over the shop: One of the subordinates assumes the position previously held by the deleted node. This promotion can cause a cascade of other promotions down the tree until a root node is left vacant and removed, or it can be stopped with other rules.

Because the adjacency list model cannot return a subtree in a single query, the constraints will have to be deferred while a traversal of some kind is performed.

2.6.1 Deleting an Entire Subtree

The simplest approach to delete an entire subtree is to do a tree traversal down from the deleted node in which you mark all of the subordinates, then go back and delete the subset of marked nodes. Let's use "–99999" as the

marker for a deleted node and defer the constraint that forbids (boss_emp_nbr = emp_nbr).

```
CREATE LOCAL TEMPORARY TABLE WorkingTable
(boss_emp_nbr INTEGER,
  emp_nbr INTEGER NOT NULL)
ON COMMIT DELETE ROWS;

CREATE PROCEDURE DeleteSubtree (IN dead_guy INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
SET CONSTRAINTS <<constraint list>> DEFERRED;
-- mark root of subtree and immediate subordinates
UPDATE OrgChart
   SET emp_nbr = CASE WHEN emp_nbr = dead_guy
                      THEN -99999 ELSE emp_nbr END,
       boss_emp_nbr = CASE WHEN boss_emp_nbr = dead_guy
                           THEN -99999 ELSE boss_emp_nbr END
WHERE dead_guy IN (emp_nbr, boss_emp_nbr);

WHILE EXISTS -- mark leaf nodes
      (SELECT *
         FROM OrgChart
        WHERE boss_emp_nbr = -99999
          AND emp_nbr > -99999)
   DO -- get list of next level subordinates

      DELETE FROM WorkingTable;
      INSERT INTO WorkingTable
      SELECT emp_nbr FROM OrgChart WHERE boss_emp_nbr = -99999;
-- mark next level of subordinates
      UPDATE OrgChart
         SET emp_nbr = -99999
       WHERE boss_emp_nbr IN (SELECT emp_nbr FROM WorkingTable);
END WHILE;

-- delete all marked nodes
DELETE FROM OrgChart
 WHERE emp_nbr = -99999;

SET CONSTRAINTS ALL IMMEDIATE;
END;
```

2.6.2 Promoting a Subordinate after Deletion

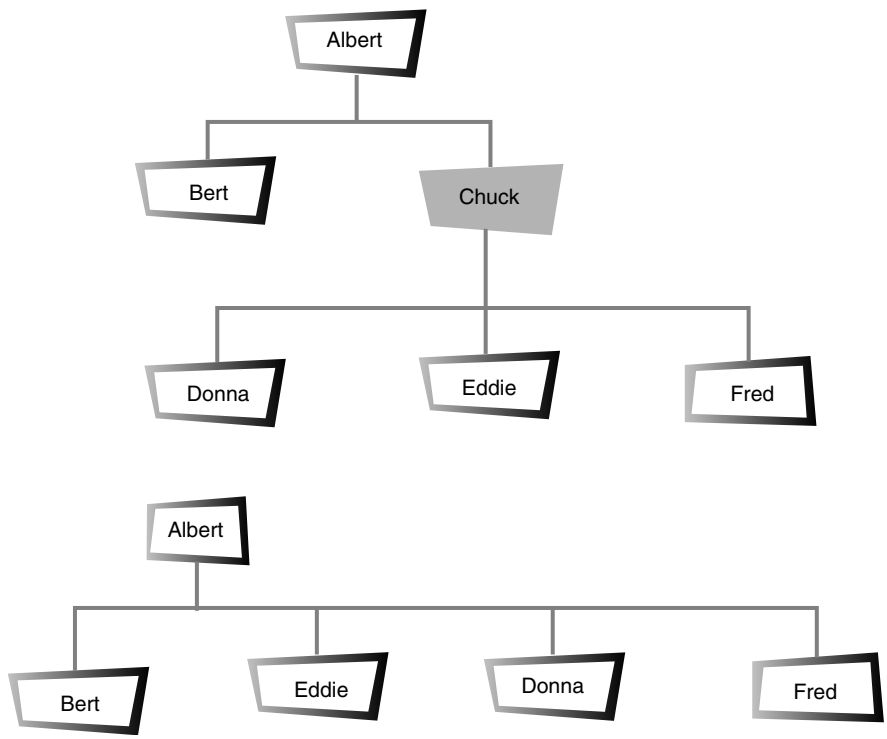
This is tricky and depends on the particular business rules. One of the more common rules is that the senior subordinate moves into the position of the deleted superior. This creates a vacancy in the old position, which might be filled by a sibling or a subordinate.

I am leaving the code to the reader, but the general idea is to rearrange the tree structure so that the dummy employee number (–9999) that we used in Section 2.6.1 is finally moved to a leaf node where it is a degenerate case of removing a subtree. For example, we could remove 'Chuck' and promote 'Donna' to his position. Her position is left vacant and can be removed, leaving 'Eddie' as the senior subordinate.

2.6.3 Promoting an Entire Subtree after Deletion

You cannot delete the root, or the tree unravels into a forest of disjointed subtrees. The constraints will prevent this from happening, but you can also test for the root in the insertion statement. Let's use the WorkingTable to hold intermediate traversal results again (Figure 2.3).

Fig. 2.3



```
CREATE PROCEDURE DeleteAndPromoteSubtree (IN dead_guy INTEGER)
LANGUAGE SQL
DETERMINISTIC
SET CONSTRAINTS <<list of constraints>> DEFERRED;
BEGIN ATOMIC
DECLARE my_emp_nbr INTEGER;
DECLARE my_boss_emp_nbr INTEGER;

INSERT INTO Workingtable (emp_nbr, boss_emp_nbr)
SELECT T1.emp_nbr, T2.boss_emp_nbr
  FROM OrgChart AS O1, OrgChart AS O2
 WHERE dead_guy IN (O1.boss_emp_nbr, O2.emp_nbr)
   AND dead_guy
       * (SELECT emp FROM OrgChart WHERE boss_emp_nbr IS NULL);

UPDATE Personnel_OrgChart

   SET boss = CASE WHEN OrgChart.boss_emp_nbr = dead_guy
                   THEN WorkingTable.emp_nbr
                   ELSE OrgChart.boss_emp_nbr END,
     emp = CASE WHEN OrgChart.emp_nbr = dead_guy
                   THEN WorkingTable.boss_emp_nbr
                   ELSE OrgChart.emp_nbr END
WHERE dead_guy IN (emp_nbr, boss_emp_nbr)
   AND dead_guy <> (SELECT emp_nbr
                      FROM OrgChart
                     WHERE boss_emp_nbr IS NULL);

DELETE FROM OrgChart
  WHERE boss_emp_nbr = emp_nbr;
END;

SET CONSTRAINTS ALL IMMEDIATE;
```

2.7 Leveled Adjacency List Model

This next approach is the result of an article Dr. David Rozenshtein wrote in the now-defunct Sybase user's *SQL FORUM* magazine (Vol. 3, No. 4, 1995). The approach he took was to do a breadth-first search, instead of a depth-first search of the tree.

Rozenshtein's objection was that processing a single node at a time leads to algorithms of complexity O(n), whereas processing the nodes by levels leads to algorithms of complexity O(log2(n)).

His model is a modified adjacency list mode, with an extra column for the level of the node in the tree. Here is a sample tree, with levels filled in. (Note: LEVEL is a reserved word in SQL-99, as well as in some SQL products.)

```
CREATE TABLE Tree
(boss CHAR(1), -- null means root
emp CHAR(1) NOT NULL,
lvl INTEGER DEFAULT 0 NOT NULL);
```

Tree

boss	emp	lvl
NULL	'a'	1
'a'	'b'	2
'a'	'c'	2
'b'	'd'	3
'b'	'e'	3
'b'	'f'	3
'e'	'g'	4
'e'	'h'	4
'f'	'i'	4
'g'	'j'	5
'i'	'k'	5
'i'	'l'	5

2.7.1 Numbering the Levels

Assigning level numbers is a simple loop, done one level at a time. Let's assume that all level numbers start as zeros.

```
CREATE PROCEDURE RenumberLevels()
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC

DECLARE lvl_counter INTEGER;
SET lvl_counter = 1;

-- set root to 1, others to zero
UPDATE Tree
   SET lvl
       = CASE WHEN boss IS NULL THEN 1 ELSE 0 END;
```

```
-- loop thru lvls of the tree
WHILE EXISTS (SELECT * FROM Tree WHERE lvl = 0)
   DO UPDATE Tree
         SET lvl = lvl_counter + 1
       WHERE (SELECT T2.lvl
                FROM Tree AS T2
               WHERE T2.emp = Tree.boss) > 0
                 AND lvl = 0;

   SET lvl = lvl_counter + 1;
END WHILE;
END;
```

The level number can be used for displaying the tree as an indented list in a host language via a cursor, but it also lets us traverse the tree by levels instead of one node at a time.

2.7.2 Aggregation in the Hierarchy

Aggregation up a hierarchy is a common form of report. Imagine that the tree is a simple parts explosion, and the weight of each assembly (root node of a subtree) is the sum of its subassemblies (all the subordinates in the subtree). The table now has an extra column for the weight, and we have information on only the leaf nodes when we start.

```
CREATE TABLE PartsExplosion
(assembly CHAR(1), -- null means root
subassembly CHAR(1) NOT NULL,
weight INTEGER DEFAULT 0 NOT NULL,
lvl INTEGER DEFAULT 0 NOT NULL);
```

I am going to create a temporary table to hold the results, and then use this table in the SET clause of an UPDATE statement to change the original table. You can actually combine these statements into a more compact form, but the code would be a bit harder to understand.

```
CREATE LOCAL TEMPORARY TABLE Summary
(node CHAR(1) NOT NULL PRIMARY KEY,
 weight INTEGER DEFAULT 0 NOT NULL)
 ON COMMIT DELETE ROWS;
CREATE PROCEDURE SummarizeWeights()
LANGUAGE SQL
```

```
DETERMINISTIC
BEGIN ATOMIC
DECLARE max_lvl INTEGER;
SET max_lvl = (SELECT MAX(lvl) FROM PartsExplosion);

-- start with leaf nodes
INSERT INTO Summary (node, total)
SELECT emp, weight
  FROM PartsExplosion
 WHERE emp NOT IN (SELECT assembly FROM PartsExplosion);

-- loop up the tree, accumulating totals
WHILE max_lvl > 1
DO INSERT INTO Summary (node, total)
   SELECT T1.assembly, SUM(S1.weight)
     FROM PartsExplosion AS T1, Summary AS S1
    WHERE T1.assembly = S1.node
      AND T1.lvl = max_lvl
    GROUP BY T1.assembly;
  SET max_lvl = max_lvl - 1;
END WHILE;

-- transfer calculations to PartsExplosion table
UPDATE PartsExplosion
   SET weight
       = (SELECT weight
            FROM Summary AS S1
           WHERE S1.node = PartsExplosion.emp)
  WHERE subassembly IN (SELECT assembly FROM PartsExplosion);
END;
```

The adjacency model leaves little choice about using procedural code because the edges of the graph are shown in single rows, without any relationship to the tree as a whole.

Path Enumeration Models

ONE OF THE properties of trees is that there is one (and only one) path from the root to every node in the tree. The path enumeration model stores that path as a string by concatenating either the edges or the keys of the nodes in the path. Searches are done with string functions and predicates on those path strings. For other references you should consult *Advanced Transact-SQL for SQL Server 2000* (Chapter 16) by Itzik Ben-Gan and Tom Moreau, (APress, Berkeley, CA; 2000; ISBN 1-893115-82-8). With this book they made the path enumeration model popular. The code in this book is product-specific, but easily generalized.

There are two methods for enumerating the paths: edge enumeration and node enumeration. The node enumeration is the most commonly used of the two methods, and there is little difference in the basic string operations on either model. However, the edge enumeration model has some numeric properties that can be useful.

It is probably a good idea to give the nodes a CHAR(n) identifier of a known size and format to make the path concatenations easier to handle. The other alternative is to use VARCHAR(n) strings, but put a separator character between each node identifier in the concatenation—a character that does not appear in the identifier itself.

To keep the examples as simple as possible, let's use my five-person Personnel_OrgChart table and a CHAR(1) identifier column to build a path enumeration model.

```
--path is a reserved word in SQL - 99
--CHECK() constraint prevents separator in the column.

CREATE TABLE Personnel_OrgChart
(emp_name CHAR(10) NOT NULL,
emp_id CHAR(1) NOT NULL PRIMARY KEY
       CHECK(REPLACE (emp_id, '/', '') = emp_id) ),
path_string VARCHAR(500) NOT NULL);
```

Personnel_OrgChart

emp_name	emp_id	path_string
'Albert'	'A'	'A'
'Bert'	'B'	'AB'
'Chuck'	'C'	'AC'
'Donna'	'D'	'ACD'
'Eddie'	'E'	'ACE'
'Fred'	'F'	'ACF'

Note that I have not broken the sample table into Personnel (emp_id, path_string) and OrgChart (emp_id, emp_name) tables. That would be a better design, but allow me this bit of sloppiness to make the code simpler to read. REPLACE (<str_exp_1>, <str_exp_2>, <str_exp_3>) is a common vendor string function. The first string expression is searched for all occurrences of the second string expression. If it is found, the second string expression is replaced by the third string expression. The third string expression can be the empty string, as in the CHECK () constraint just given.

Another problem is how to prevent cycles in the graph. A cycle would be represented as a path string in which at least one emp_id string appears twice, such as 'ABCA' in my sample table. This can be done with a constraint that uses a subquery, thus:

```
CHECK (NOT EXISTS
      (SELECT *
         FROM Personnel_OrgChart AS D1,
              Personnel_OrgChart AS P1
        WHERE CHAR_LENGTH (REPLACE (D1.emp_id, P1.path_string, ''))
              < (CHAR_LENGTH(P1.path_string)
                 - 1) --size of one emp_id string
      ) )
```

Another fact about such a tree is that no path can be longer than the number of nodes in the tree.

```
CHECK ((SELECT MAX(CHAR_LENGTH(path_string) )
           FROM Personnel_OrgChart AS P1)
        <= (SELECT COUNT(emp_id) * CHAR_LENGTH(emp_id)
              FROM Personnel_OrgChart AS P2))
```

This assumes that the emp_id is of fixed length and that no separators were used between them in the path_string. Unfortunately, the SQL-92 feature of a subquery in a constraint is not widely implemented yet.

3.1 Finding the Depth of the Tree

If you have used fixed length emp_id string, then the depth is the length of the path, divided by the length of the emp_id string, CHAR_LENGTH(emp_id).

```
CHAR_LENGTH(path_string) / CHAR_LENGTH(emp_id)
```

I made it easy to compute by using a single character emp_id code. This is not usually possible in a real tree, with several hundred nodes.

If you used a varying length emp_id, then the depth is:

```
CHAR_LENGTH(path_string) - CHAR_LENGTH (REPLACE (path_string,
'/', '') ) + 1
```

As explained earlier in this chapter, the REPLACE() function is not a Standard SQL string function; however, it is quite common in actual SQL products. This approach counts the separators.

3.2 Searching for Subordinates

Given a parent, find all of the subtrees under it. The immediate solution is as follows:

```
SELECT *
  FROM Personnel_OrgChart
 WHERE path_string LIKE '%' || :parent_emp_id || '%';
```

The problem is that searches with LIKE predicates whose patterns begin with a '%' wildcard are slow. This is because they usually generate a table scan. Also, note that using '_%' in front of the LIKE predicate pattern will exclude the root of the subtree from the answer. Another approach is to use the following query:

```
SELECT *
  FROM Personnel_OrgChart
```

```
 WHERE path_string
       LIKE (SELECT path_string
               FROM Personnel_OrgChart
              WHERE emp_id = :parent_emp_id) || '%';
```

The subquery will use the indexing on the emp_id column to find the "front part" of the path string, from the root to the parent with whom we are concerned.

Traveling down the tree is easy. Instead of '%' wildcard, use a string of underscore ('_') wildcards of the right length. For example, this will find the immediate children of a given parent emp_id.

```
SELECT *
  FROM Personnel_OrgChart
 WHERE path_string
       LIKE (SELECT path_string
               FROM Personnel_OrgChart
              WHERE emp_id = :parent_emp_id) ||'_';
```

Many SQL products have a function that will pad a string with repeated copies of an input string or return a string of repeated copies of an input string. For example, SQL Server has REPLICATE (<character exp>, <integer exp>) and Oracle has LPAD() and RPAD(). This can be useful for generating a search pattern of underscores on the fly.

```
SELECT *
  FROM Personnel_OrgChart
 WHERE path_string
       LIKE (SELECT path_string
               FROM Personnel_OrgChart
              WHERE emp_id = :parent_emp_id) || REPLICATE
              ('_', :n);
```

3.3 Searching for Superiors

Given a node, find all of its superiors. This requires disassembling the path back into the identifiers that constructed it. We can use a table of sequential integers to find the required substrings:

```
SELECT SUBSTRING (P1.path_string
                  FROM (seq * CHAR_LENGTH(P1.emp_id) )
                  FOR CHAR_LENGTH(P1.emp_id) ) AS emp_id
```

```
  FROM Personnel_OrgChart AS P1, Sequence AS S1
  WHERE P1.emp_id = :search_emp_id
    AND S1.seq <= CHAR_LENGTH(path_string)/CHAR_LENGTH(emp_id);
```

The problem is that this does not tell you the relationships among the superiors, only who they are. Those relationships are actually easier to report.

```
SELECT P2.*
  FROM Personnel_OrgChart AS P1,
       Personnel_OrgChart AS P2
 WHERE P1.emp_id = :search_emp_id
   AND POSITION (P2.path_string IN P1.path_string) = 1;
```

3.4 Deleting a Subtree

Given a node, delete the subtree rooted at that node. This can be done with the same predicate as finding the subordinates:

```
DELETE FROM Personnel_OrgChart
  WHERE path_string
        LIKE (SELECT path_string
                FROM Personnel_OrgChart
               WHERE emp_id = :dead_guy) || '%';
```

3.5 Deleting a Single Node

Once more we have to face the problem that when a nonleaf node is removed from a tree, it is no longer a tree and we need to have rules for changing the structure.

If we simply move everyone up a level in the tree, we can first remove that node emp_id from the Personnel_OrgChart table, and then remove that emp_id from the paths of the other nodes.

```
BEGIN ATOMIC
DELETE FROM Personnel_OrgChart
 WHERE emp_id = :dead_guy;
UPDATE Personnel_OrgChart
   SET path_string = REPLACE (path_string, :dead_guy, '');
END;
```

There are other methods of rebuilding the tree structure after a node is deleted, as we have discussed. Promoting a subordinate based on some criteria

to the newly vacant position and leaving a vacancy in the organizational chart, and so forth, are all options. These methods are usually implemented with some combination of node deletions and insertions.

3.6 Inserting a New Node

The enumeration model has the same insertion properties as the adjacency list model. The new emp_id is simply concatenated to the end of the path of the parent node to which it is subordinated.

```
INSERT INTO Personnel_OrgChart
VALUES (:new_guy, :new_emp_id,
       (SELECT path_string FROM Personnel_OrgChart WHERE emp_id
= :new_guy_boss)
       || :new_emp_id);
```

This basic statement design can be modified to work for insertion of a subtree, thus:

```
INSERT INTO Personnel_OrgChart
SELECT N1.emp, N1.emp_id,
        (SELECT path_string FROM Personnel_OrgChart WHERE emp_id
= :new_tree_boss)
        || N1.emp_id
 FROM NewTree AS N1;
```

3.7 Splitting up a Path String

The path string contains information about the nodes in the path it represents, so you will often want to split it back into the nodes that are created. This is easier to do if the path string was built with a separator character, such as a comma or slash; I will use a slash, so this will look like a directory path in UNIX. You will also need a table called "Sequence," which is a set of integers from 1 to (n).

CharIndex(<search string>, <target string>, <starting position>) is a vendor version of the Standard SQL function POSITION(<search string> IN <target string>). It begins the search at a position in the target string, thus when the <starting position> = 1, the two are equivalent. It can be defined as:

```
CREATE FUNCTION CharIndex (IN search_str VARCHAR(1000), IN
target VARCHAR(1000), IN start_point INTEGER) RETURNS INTEGER
RETURN
```

```
   (POSITION (search_str
              IN SUBSTRING (target FROM start_point) ) +
start_point - 1);
```

Version one:

```
SELECT CASE WHEN SUBSTRING('/' || P1.path_string || '/' FROM
S1.seq FOR 1) = '/'
            THEN SUBSTRING('/' || P1.path_string || '/' FROM
(S1.seq + 1)
                           FOR CharIndex('/', '/' ||
P1.path_string || '/', S1.seq + 1)
                                  - S1.seq - 1)
            ELSE NULL END AS emp_id
 FROM Sequence AS S1, Personnel_OrgChart AS P1
WHERE S1.seq BETWEEN 1 AND CHAR_LENGTH('/' || P1.path_string ||
'/') - 1
  AND SUBSTRING('/' || P1.path_string || '/' FROM S1.seq FOR 1) =
'/'
```

Version two: This version uses the same idea, but with two sequence numbers to bracket the emp_id embedded in the path string. It also returns the position of the subordinate emp_id in the path.

```
CREATE VIEW Breakdown (emp_id, step_nbr, subordinate_emp_id)
AS
SELECT emp_id,
   COUNT(S2.seq),
   SUBSTRING('/' || P1.path_string ||  '/' FROM MAX(S1.seq + 1)
        FOR S2.seq - MAX(S1.seq + 1))
 FROM Personnel_OrgChart AS P1, Sequence AS S1, Sequence AS S2
WHERE SUBSTRING('/' ||  P1.path_string || '/' FROM S1.seq FOR 1)
= '/'
  AND SUBSTRING('/' || P1.path_string || '/' FROM S2.seq FOR 1) =
'/'
 AND S1.seq < S2.seq
 AND S2.seq <= CHAR_LENGTH(P1.path_string) + 1
GROUP BY P1.emp_id, P1.path_string, S2.seq;
```

The S1 and S2 copies of Sequence are used to locate bracketing pairs of separators, and the entire set of substrings located between them is extracted in one step. The trick is to be sure that the left-hand separator of the bracketing

pair is closest to the second separator. The step_nbr column tells you the relative position of the subordinate employee to the employee in the path.

Version three: This version is the same as version 2, but is more concise and easier to comprehend.

```
SELECT SUBSTRING('/' || P1.path_string || '/'
                 FROM S1.seq + 1
                 FOR CharIndex('/',
                               '/' || P1.path_string || '/',
                               S1.seq +1) - S1.seq - 1) AS node
  FROM Sequence AS S1, Personnel_OrgChart AS P1
 WHERE SUBSTRING('/' || P1.path_string || '/'
                 FROM S1.seq FOR 1) = '/'
   AND seq < CHAR_LENGTH('/' || P1.path_string || '/');
```

Version four: This version shows another way of using the LIKE predicate:

```
SELECT SUBSTRING(P1.path_string
                 FROM seq +1
                  FOR CharIndex('/', P1.path_string, S1.seq +  1)
- (S1.seq + 1) )
  FROM Sequence AS S1
       INNER JOIN
       (SELECT '/' || path_string || '/'
          FROM Personnel_OrgChart) AS P1.(path_string)
       ON S1.seq <= CHAR_LENGTH(P1.path_string)
           AND SUBSTRING(P1.path_string
                         FROM S1.seq
                         FOR CHAR_LENGTH(P1.path_string) )
              LIKE '/_%';
```

3.8 The Edge Enumeration Model

So far we have seen the node enumeration version of the path enumeration model. In the edge enumeration model the "driving directions" for following the path from the root to each node are given as integers. You will also recognize it as the way that the book you are reading is organized. The path column contains a string of the edges that make up a path from the root ('King') to each node, numbering them from left to right at each level in the tree.

Personnel_OrgChart

emp_name	edge_path
'Albert'	'1.'
'Bert'	'1.1.'
'Chuck'	'1.2.'
'Donna'	'1.2.1.'
'Eddie'	'1.2.2.'
'Fred'	'1.2.3.'

For example, 'Donna' is the second child of the first child ('Chuck') of the root ('Albert'). This assigns a partial ordering to the nodes of the trees. The main advantage of this notation is that you do not have to worry about long strings; however, there is no real difference in the manipulations. The numbering does give an implied ordering to siblings that might have meaning.

3.9 XPath and XML

I have avoided mentioning XML because this is a book on SQL, but I cannot avoid it forever because the two are becoming more and more linked. XML is a mark-up language that shows a data element hierarchy by inserting tags into the text file, which holds the data elements.

XML is becoming the "Esperanto" for moving data from one source to another, and there are many tools that are de jure, or de facto, standards for doing queries on the data while it is in XML. One of these tools is XPath, which is based on a fairly simple notation to describe paths to nodes in an XML document in a notation that resembles a path enumeration, but with wildcards and other higher-level features.

The nodes on the path can then be sent as input to functions. Older programmers can think of XPath as a nonprocedural version of IMS or other hierarchical database query languages.

DevelopMentor (http://develop.com/us/default.aspx) offers free tutorials on XML online. If you would like to play with XPath queries, go to http://staff.develop.com/aarons/bits/xpath-expression-builder-4.0/, which belongs to Aaron Skonnard. You will find a tool that allows you to run XPath queries and functions against an XML file and see the results graphically.

This page intentionally left blank

Nested Set Model of Hierarchies

TREES ARE OFTEN drawn as "boxes-and-arrows" charts, which tend to lock your mental image of a tree into a graph structure. Another way of representing trees is to show them as nested sets. It is strange that this approach was overlooked for so long among SQL programmers. Many of us are old enough to have used *The Art of Computer Programming* (Donald E. Knuth, ed 3, vol 1, Boston, Addison-Wesley, 1997) in college as our textbook, and we should remember this representation of trees on page 312, Figure 20.

Because SQL is a set-oriented language, this is a better model for the approach discussed here. Let us define an organizational chart table to represent the hierarchy and people in our sample organization. The first column is the name of the member of this organization. I will explain the (lft-rgt) columns shortly, but for now note that their names are abbreviations for LEFT and RIGHT which are reserved words in SQL-92.

```
CREATE TABLE OrgChart
(member CHAR(10) NOT NULL PRIMARY KEY,
lft INTEGER NOT NULL,
rgt INTEGER NOT NULL);
```

OrgChart

member	lft	rgt
'Albert'	1	12
'Bert'	2	3
'Chuck'	4	11
'Donna'	5	6
'Eddie'	7	8
'Fred'	9	10

To show a tree as nested sets, replace the boxes with ovals and then nest subordinate ovals inside their parents. Containment represents subordination. The root will be the largest oval and will contain every other node. The leaf nodes will be the innermost ovals, with nothing else inside them, and the nesting will show the hierarchical relationship. This is a natural way to model a parts explosion because a final assembly is made of physically nested assemblies that finally break down into separate parts (Figure 4.1). This tree translates into a nesting of sets, as illustrated in Figure 4.2. Using this approach, we can model a tree with (lft, rgt) nested sets with number pairs. These number pairs will always contain the pairs of their subordinates, so that a child node is within the bounds of its parent. Figure 4.3 shows a version of the nested sets, flattened onto a number line.

Fig. 4.1

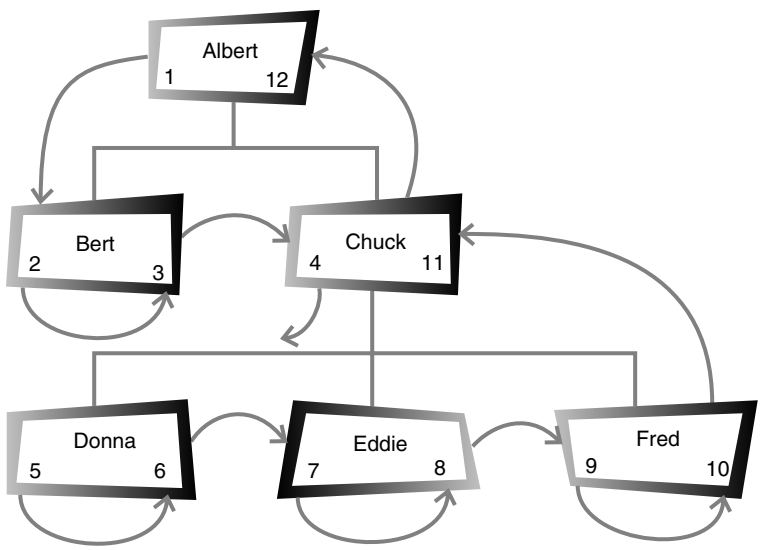
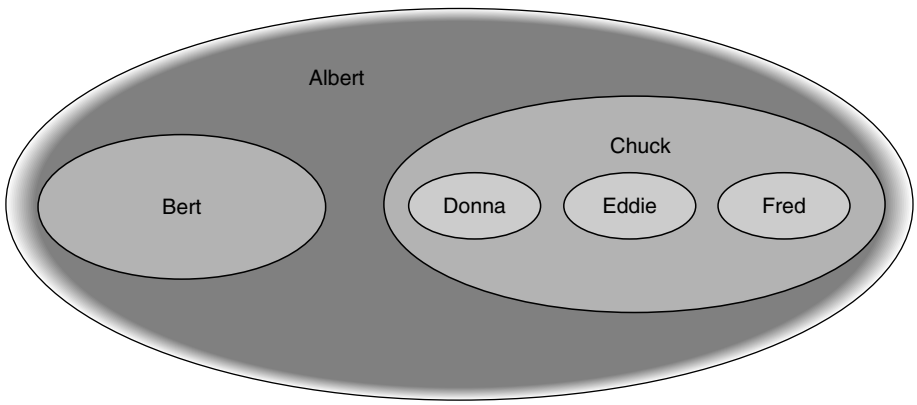
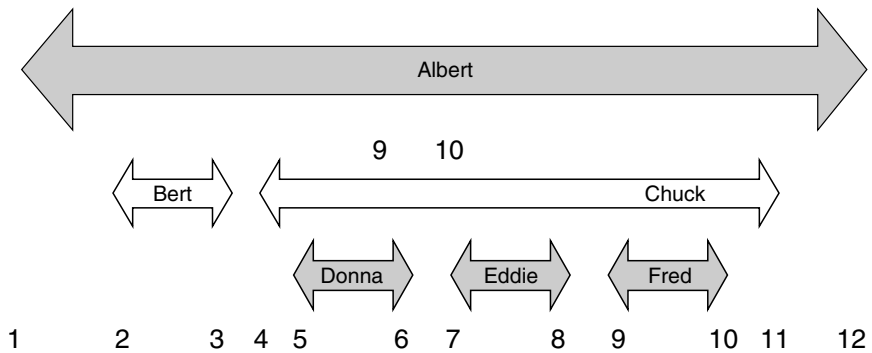


Fig. 4.2



As nested circles

Fig. 4.3



As intervals on a number line

If that mental model does not work for you, visualize the nested sets model as a little worm with a Bates automatic numbering stamp crawling along the "boxes-and-arrows" version of the tree. The worm starts at the top, the root, and makes a complete trip around the tree. When he comes to a node, he puts a number in the cell on the side that he is visiting and his numbering stamp increments itself. Each node will get two numbers, one for the rgt side and one for the lft side. Computer science majors will recognize this as a preorder (or depth-first) tree traversal algorithm with a modification for numbering the nodes. This numbering has some predictable results that we can use for building queries.

4.1 Finding Root and Leaf Nodes

The root will always have a 1 in its lft column and twice the number of nodes in its rgt column. This is easy to understand; the worm has to visit each node twice, once for the lft side and once for the rgt side, so the final count has to be twice the number of nodes in the whole tree. The root of the tree is found with the following query:

```
SELECT *
  FROM Orgchart
WHERE lft = 1;
```

This query will take advantage of an index on the left (lft) value. A leaf node is one that has no children under it. In an adjacency matrix model it is not that easy to find all the leaf nodes because you have to use a correlated subquery:

```
SELECT *
  FROM OrgChart AS O1
WHERE NOT EXISTS
   (SELECT *
      FROM OrgChart AS O2
     WHERE O1.member = O2.boss);
```

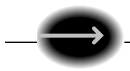
In the nested sets table the difference between the (lft, rgt) values of leaf nodes is always 1. Think of the little worm turning the corner as he crawls along the tree. That means you can find all leaf nodes with the following extremely simple query:

```
SELECT *
  FROM Orgchart
WHERE(rgt - lft) = 1;
```

There is a further trick to speed up queries. Build a unique index on either the lft column or on the pair of columns (lft, rgt), and then the optimizer can use just the index instead of searching the base table itself.

```
SELECT *
  FROM Orgchart
WHERE lft =(rgt - 1);
```

The reason this improves performance is that the SQL engine can use an index on the lft column when it does not appear in an expression. Don't use (rgt - lft) = 1 because it will prevent the index from being used.



4.2 Finding Subtrees

Trees have a lot of special properties, and those properties are very useful to us. A tree is a graph that has no cycles in it; that is, no path folds back on itself to catch you in an endless loop when you follow it. Another defining property is that there is always a path from the root to any other node in the tree.

Another useful property is that any node in the tree is the root of a subtree, and certain properties of that subtree are immediately available from the (lft, rgt) pair. In the nested sets table all the descendants of a node can be found by looking for the nodes whose (lft, rgt) numbers are between the (lft, rgt) values of their parent node. This is the nesting expressed in number ranges instead of in a drawing of circles within circles. For example, to find out all subordinates of each boss in the organizational hierarchy you would write:

```
SELECT Mgrs.member AS boss, Workers.member AS worker
  FROM Orgchart AS Mgrs, Orgchart AS Workers
WHERE Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
  AND Workers.rgt BETWEEN Mgrs.lft AND Mgrs.rgt;
```

Look at the way the numbering was done and you can convince yourself that this search condition is too strict. We can drop the last predicate and simply use:

```
SELECT Mgrs.member AS boss, Workers.member AS worker
  FROM Orgchart AS Mgrs, Orgchart AS Workers
WHERE Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt;
```

This would tell you that everyone is also his own superior, so in some situations you would also add the predicate

```
.. AND Workers.lft <> Mgrs.lft
```

or change it to

```
WHERE Workers.lft > Mgrs.lft
  AND Workers.lft < Mgrs.rgt;
```

This simple self-JOIN query is the basis for almost everything that follows in the nested sets model.

4.3 Finding Levels and Paths in a Tree

The level of a node in a tree is the number of edges between the node and the root, where the larger the depth number, the farther away the node is from the root. A path is a set of edges that directly connect two nodes.

The nested sets model uses the fact that each containing set is "wider" (where width = [rgt - lft]) than the sets it contains. Obviously the root will always be the widest row in the table. The level function is the number of edges between two given nodes; it is easy to calculate. For example, to find the level of each worker you would use:

```
SELECT O2.member, COUNT(O1.member) AS level
  FROM OrgChart AS O1, OrgChart AS O2
WHERE O2.lft BETWEEN O1.lft AND O1.rgt
GROUP BY O2.member;
```

The expression COUNT(01.member) will count the node itself; if you prefer to start at zero, use (COUNT(01.member) - 1). You will see it done both ways in the literature.

4.3.1 Finding the Height of a Tree

The height of a tree is the length of the longest path in the tree. We know that this path runs from the root to a leaf node, so we can write the following query to find the height:

```
SELECT MAX(level) AS height
  FROM (SELECT O2.member, (COUNT(O1.member) - 1)
          FROM OrgChart AS O1, OrgChart AS O2
         WHERE O2.lft BETWEEN O1.lft AND O1.rgt
         GROUP BY O2.member) AS L1(member, level);
```

Other queries can be built from this tabular subquery expression of the nodes and their level numbers. If you find yourself using this subquery expression often, you might consider creating a VIEW from this expression.

4.3.2 Finding Levels of Subordinates

The adjacency model allows you to find the immediate subordinates of a node quickly; you simply look in the columns that provide the parent of each child of each node in the tree. The real problem is finding a given generation or level in the tree.

This becomes complicated in the nested set model. The immediate subordinates are defined then as personnel who have no other employee between themselves and their boss.

```
CREATE VIEW Immediate_Subordinates(boss, worker, lft, rgt)
AS SELECT Mgrs.member, Workers.member, Workers.lft, Workers.rgt
     FROM OrgChart AS Mgrs, OrgChart AS Workers
   WHERE Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
     AND NOT EXISTS -- no middle manager between the boss and us!
       (SELECT *
         FROM OrgChart AS MidMgr
        WHERE MidMgr.lft BETWEEN Mgrs.lft AND Mgrs.rgt
          AND Workers.lft BETWEEN MidMgr.lft AND MidMgr.rgt
          AND MidMgr.member NOT IN(Workers.member, Mgrs.member));
```

You also need to look at Section 4.9 (Converting Nested Sets Model to Adjacency List) for better answers regarding immediate subordinates. I am simply giving an elaborate query here to show a pattern. Likewise, Mgrs.member can be replaced with Workers.boss in the SELECT statement.

There is a reason for setting this up as a VIEW and including the (lft, rgt) numbers of the children. The (lft, rgt) numbers for the parent of each node can be reconstructed by:

```
SELECT boss, MIN(lft) - 1, MAX(rgt) + 1
  FROM Immediate_Subordinates
  GROUP BY boss;
```

This query can be generalized to any distance (:n) in the hierarchy, thus:

```
SELECT Workers.member, 'is', :n, 'levels down from',
:my_member
  FROM OrgChart AS Mgrs, OrgChart AS Workers
 WHERE Mgrs.member = :my_member
  AND Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
 AND :n = (SELECT COUNT(MidMgr.member) + 1
             FROM OrgChart AS MidMgr
            WHERE MidMgr.lft BETWEEN Mgrs.lft AND Mgrs.rgt
              AND Workers.lft BETWEEN MidMgr.lft AND MidMgr.rgt
              AND MidMgr.member
                  NOT IN(Workers.member, Mgrs.member));
```

This query can be flattened out, and it probably would run faster without the subquery:

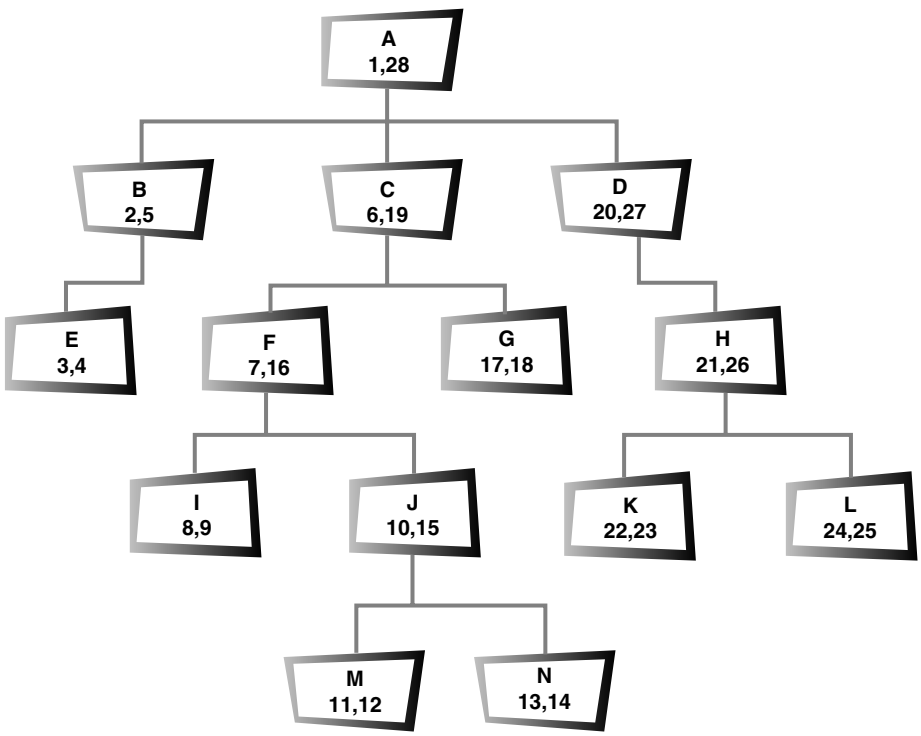
```
SELECT Workers.member, 'is', :n, 'levels down from',
:my_member
  FROM OrgChart AS Mgrs, OrgChart AS Workers,
       OrgChart AS MidMgr
 WHERE Mgrs.member = :my_member
   AND Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
   AND MidMgr.lft BETWEEN Mgrs.lft AND Mgrs.rgt
   AND Workers.lft BETWEEN MidMgr.lft AND MidMgr.rgt
   AND MidMgr.member NOT IN(Workers.member, Mgrs.member)
 GROUP BY Workers.member
HAVING :n = COUNT(MidMgr.member);
```

In the nested sets model, queries based on subtrees are usually easier to write than those for individual nodes or other subsets of the tree.

Switching to another hierarchy, let's look at a simple parts explosion (Figure 4.4). This table will be modified in later examples to include more information, but for now just assume that it looks like this:

```
CREATE TABLE Assemblies
(part CHAR(2) NOT NULL
       REFERENCES Inventory(part) -- assume an inventory
       ON UPDATE CASCADE,
lft INTEGER NOT NULL,
rgt INTEGER NOT NULL,
...);

INSERT INTO Assemblies
VALUES ('A', 1, 28),
       ('B', 2, 5),
       ('C', 6, 19),
       ('D', 20, 27),
       ('E', 3, 4),
       ('F', 7, 16),
       ('G', 17, 18),
       ('H', 21, 26),
       ('I', 8, 9),
       ('J', 10, 15),
       ('K', 22, 23),
       ('L', 24, 25),
       ('M', 11, 12),
       ('N', 13, 14);
```

Fig. 4.4



If you want, show the levels as a single row, where NULLs are used to show that there is no part at that level.

```
CREATE VIEW Flat_Parts(part, level_0, level_1, level_2, level_3)
AS
SELECT A1.part,
       CASE WHEN COUNT(A3.part) = 2
            THEN A2.node
            ELSE NULL END AS level_0,
       CASE WHEN COUNT(A3.part) = 3
            THEN A2.node
            ELSE NULL END AS level_1,
       CASE WHEN COUNT(A3.part) = 4
            THEN A2.part
            ELSE NULL END AS level_2,
       CASE WHEN COUNT(A3.part) = 5
```

```
            THEN A2.part
            ELSE NULL END AS level_3
 FROM Assemblies AS A1, -- subordinates
      Assemblies AS A2, -- superiors
      Assemblies AS A3, -- items in between them
WHERE A1.lft BETWEEN A2.lft AND A2.rgt
  AND A3.lft BETWEEN A2.lft AND A2.rgt
  AND A1.lft BETWEEN A3.lft AND A3.rgt
GROUP BY A1.part, A2.part;
```

Now you can write a query to show the path from a node to the root of the tree horizontally.

```
SELECT part, MAX(level_0), MAX(level_1),
             MAX(level_2), MAX(level_3)
  FROM Flat_Parts
 GROUP BY part;
```

You could also fold all of this into one query, but the VIEW is useful for other reports. Another way to flatten the tree is credited to Richard Romley of Smith-Barney in New York. He claims that the following query runs with half the I/O of the VIEW-based solution in SQL Server:

```
SELECT A1.part,
       (SELECT part
          FROM Assemblies
         WHERE lft = MAX(A2.lft)) AS level_0,
       (SELECT part
          FROM Assemblies
         WHERE lft = MAX(A3.lft)) AS level_1,
       (SELECT part
          FROM Assemblies
         WHERE lft = MAX(A4.lft)) AS level_2,
       (SELECT part
          FROM Assemblies
         WHERE lft = MAX(A5.lft)) AS level_3
FROM Assemblies AS A1
     LEFT OUTER JOIN
     Assemblies AS A2
     ON A1.lft > A2.lft AND A1.rgt < A2.rgt
```

```
          LEFT OUTER JOIN
          Assemblies AS A3
          ON A2.lft > A3.lft AND A2.rgt < A3.rgt
             LEFT OUTER JOIN
             Assemblies AS A4
             ON A3.lft > A4.lft AND A3.rgt < A4.rgt
                LEFT OUTER JOIN
                Assemblies AS A5
                ON A4.lft > A5.lft AND A4.rgt < A5.rgt
GROUP BY A1.part;
```

This query is a little tricky on two points. The use of an aggregate in a WHERE clause is generally not allowed, but the MAX() is an outer reference in the scalar subqueries; therefore it is valid Standard SQL-92. The nested LEFT OUTER JOINs reflect the nesting of the (lft, rgt) ranges, but they will return NULLs when there is nothing at a particular level.

The result is:

Result

part	level_0	level_1	level_2	level_3
'A'	NULL	NULL	NULL	NULL
'B'	'A'	NULL	NULL	NULL
'C'	'A'	NULL	NULL	NULL
'D'	'A'	NULL	NULL	NULL
'E'	'B'	'A'	NULL	NULL
'F'	'C'	'A'	NULL	NULL
'G'	'C'	'A'	NULL	NULL
'H'	'D'	'A'	NULL	NULL
'I'	'F'	'C'	'A'	NULL
'J'	'F'	'C'	'A'	NULL
'K'	'H'	'D'	'A'	NULL
'L'	'H'	'D'	'A'	NULL
'M'	'J'	'F'	'C'	'A'
'N'	'J'	'F'	'C'	'A'

Both approaches are compact, easy to follow, and easy to expand to as many levels as desired.

4.3.3 Finding Oldest and Youngest Subordinates

The nested sets model usually assumes that the subordinates are ranked by age, seniority, or in some way from left to right among the immediate subordinates of a node. The adjacency model does not have a concept of such rankings, so the following queries are not possible without extra columns to hold the rankings in the adjacency list model.

Most senior subordinates are found by the following query:

```
SELECT Workers.member, 'is the most senior subordinate of',
:my_member
  FROM OrgChart AS Mgrs, OrgChart AS Workers
 WHERE Mgrs.member = :my_member
  AND Workers.lft = Mgrs.lft + 1; -- leftmost child
```

Most junior subordinates are found by the following query:

```
SELECT Workers.member, 'is the least senior subordinate of',
:my_member
  FROM OrgChart AS Mgrs, OrgChart AS Workers
 WHERE Mgrs.member = :my_member
   AND Workers.rgt = Mgrs.rgt - 1; -- rightmost child
```

The real trick is to find the nth sibling of a parent in a tree. If you remember the old Charlie Chan movies, Detective Chan always referred to his sons by number, such as "Number One son," "Number Two son," and so forth. This becomes a self-JOIN on the set of immediate subordinates of the parent under consideration. That is why I created a VIEW for telling us the immediate subordinates before introducing this problem. The query is much easier to read using the VIEW.

```
SELECT S1.worker, 'is the', :n, '-th subordinate of',
S1.boss
  FROM Immediate_Subordinates AS S1
 WHERE S1.boss = :my_member
   AND :n = (SELECT COUNT(S2.lft) - 1
               FROM Immediate_Subordinates AS S2
              WHERE S2.boss = S1.boss
                AND S2.boss <> S1.worker
                AND S2.lft BETWEEN 1 AND S1.lft);
```

Notice that you have to subtract one to avoid counting the parent as his own child. Here is another way to do this and get a complete ordered listing of siblings:

```
SELECT O1.member AS boss, S1.worker,
       COUNT(S2.lft) AS sibling_order
  FROM Immediate_Subordinates AS S1,
       Immediate_Subordinates AS S2,
       OrgChart AS O1
 WHERE S1.boss = O1.member
   AND S2.boss = S1.boss
   AND S1.worker <> S2.worker
   AND S2.lft <= S1.lft
 GROUP BY O1.member, S1.worker;
```

The siblings of a given node can be found by looking for a common parent and rows on the same level. Using the Assemblies parts explosion tree, we can define a VIEW with the level number in it as:

```
CREATE VIEW Siblings (lvl, part, lft, rgt)
AS SELECT COUNT(A2.lft), A1.part, A1.lft, A1.rgt
     FROM Assemblies AS A1, Assemblies AS A2
    WHERE A1.lft BETWEEN A2.lft AND A2.rgt
    GROUP BY A1.part, A1.lft, A1.rgt;
```

This VIEW can then be used for

```
SELECT DISTINCT S2.part
  FROM Siblings AS S1, Siblings AS S2
 WHERE S1.part = :my_sibling_part
   AND EXISTS
       (SELECT *
          FROM Siblings AS S0
         WHERE S1.lft BETWEEN S0.lft AND S0.rgt
           AND S2.lft BETWEEN S0.lft AND S0.rgt
           AND S0.lvl = S1.lvl - 1
           AND A1.lvl = A2.lvl);
```

This query look at the parent of your current node (part), then finds all the immediate children of the parent node; these children are your siblings.

4.3.4 Finding a Path

To find and number the nodes in the path from a :start_node to a :finish_node you can repeat the nested set "BETWEEN predicate trick" twice to form an upper and a lower boundary on the set.

```
SELECT A2.part,
       (SELECT COUNT(*)
          FROM Assemblies AS A4
         WHERE A4.lft BETWEEN A1.lft AND A1.rgt
           AND A2.lft BETWEEN A4.lft AND A4.rgt) AS path_nbr

 FROM Assemblies AS A1, Assemblies AS A2, Assemblies AS A3
WHERE A1.part = :start_node
  AND A3.part = :finish_node
  AND A2.lft BETWEEN A1.lft AND A1.rgt
  AND A3.lft BETWEEN A2.lft AND A2.rgt;
```

Using the Assemblies parts explosion tree, this query would return the following table for the path from 'C' to 'N', with 1 being the highest starting node and the other nodes numbered in the order they must be traversed.

node	path_nbr
C	1
F	2
J	3
N	4

However, if you just need a column to use in a sort for output to a host language, then replace the subquery expression with "(A2.rgt-A2.lft) AS sort_col" and use an "ORDER BY sort_col" clause in a cursor.

4.3.5 Finding Relative Position

Given two nodes, can you find their relative position in the hierarchy; that is, who is the subordinate of whom or are they in different subtrees of the hierarchy?

```
SELECT CASE WHEN :first_member = : second_member
            THEN :first_member || 'is' || :second_member
            WHEN O1.lft BETWEEN O2.lft AND O2.rgt
            THEN :first_member || 'subordinate to' ||
            :second_member
```

```
            WHEN O2.lft BETWEEN O1.lft AND O1.rgt
            THEN :second_member || 'subordinate to' ||
            :first_member
            ELSE :first_member || 'no relation to' ||
            :second_member
       END
  FROM OrgChart AS O1, OrgChart AS O2
 WHERE O1.member = :first_member
   AND O2.member = :second_member;
```

This query will report all the cases, so if the same member holds various positions in the organizational chart, several rows can be returned. It also will report no relationship if one or both of the parameters is not in the table at all.

4.4 Functions in the Nested Sets Model

The level of a given node is a matter of counting how many (lft, rgt) groupings (superiors) this node's lft or rgt is within. You can get this by modifying the sense of the BETWEEN predicate in the query for subtrees:

```
SELECT :my_member, COUNT(Mgrs.member) AS level
  FROM OrgChart AS Mgrs, OrgChart AS Workers
 WHERE Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
   AND Workers.member = :my_member;
```

Let's assume that this organization is involved in a pyramid sales operation and that a supervising member gets credit for the total sales of himself and all his subordinates. First, we need to have a table for the sales that each member made.

```
CREATE TABLE Sales
(member CHAR(10) NOT NULL PRIMARY KEY,
 sale_amt DECIMAL(12,4) NOT NULL);

SELECT :my_member, SUM(S1.sale_amt) AS total_sales
  FROM OrgChart AS Mgrs, OrgChart AS Workers,
       Sales AS S1
 WHERE Workers.lft BETWEEN Mgrs.lft AND Mgrs.rgt
   AND P1.job_title = Workers.job_title
   AND Mgrs.member = :my_member;
```

A slightly trickier function involves using quantity columns in the nodes to compute an accumulated total. This usually occurs in parts explosions, where one assembly may contain several occurrences of subassemblies. Let's assume we have a table called "Blueprint," with the price and quantity for each part required for each subassembly; for example, an assembly might require 10 number 5 machine screws at $0.07 each. The total cost of any given part would be:

```
SELECT :this_part, SUM(Subassem.qty * Subassem.price) AS
totalcost
  FROM Blueprint AS Assembly, Blueprint AS Subassem
 WHERE Subassem.lft
       BETWEEN Assembly.lft AND Assembly.rgt
   AND Assembly.part = :this_part;
```

The use of AVG(), MIN(), and MAX() aggregate functions are possible, but you have to watch out for the meaning of the results in the context of your data model.

4.5 Deleting Nodes and Subtrees

Another interesting property of the nested sets model is that the subtrees must fill from lft to rgt. In other tree representations it is possible for a parent node to have a right child and no left child; however, this can make traversals more complicated in exchange for being able to assign significance to the position of a node within a group of siblings.

Deleting a single node in the middle of the tree is conceptually harder than removing whole subtrees in the nested sets model. When you remove a node in the middle of the tree, you have to decide how to fill the hole. There are several basic ways. The first method is to connect the children to the parent of the original node—Mom dies and Grandma adopts the kids. In effect the position itself is removed. This is a vertical promotion of an entire subtree.

Another vertical promotion is to move only a single child node to the deleted node's position—give the business to the oldest son. The problem is that when the son is promoted, this leaves a vacancy in his former position.

The second method is horizontal promotion. The sibling to the deleted node's right (i.e., next most senior) moves over to the vacant position—Dad dies and his oldest brother takes over the business. This assumes that there is such a brother to take the vacant position.

In practice you will find a mixture of these methods as vacancies are created in the hierarchy and have to be handled. As I said, single-node deletion is not easy.

A website with a demonstration program for the nested sets model in PHP, written by Arne Klempert (email: arne@klempert.de), can be found at http://www.klempert.de/php/nested_sets/demo/. It is under the terms of the GNU Lesser General Public License. The demo allows you add nodes or to delete nodes or subtrees with a simple interface.

4.5.1 Deleting Subtrees

This query will take the downsized employee as a parameter and remove the subtree rooted under him. The trick in this query is that we are using the node value, but we need to get the (lft, rgt) values to do the work. The answer is scalar subqueries:

```
DELETE FROM OrgChart
 WHERE lft BETWEEN (SELECT lft
                      FROM OrgChart
                      WHERE member = :downsized_guy)
                AND
                   (SELECT rgt
                      FROM OrgChart
                      WHERE member = :downsized_guy);
```

The problem is that this will result in gaps in the sequence of nested set numbers. You can still do most tree queries on a table with such gaps, but you will lose the algebraic properties that let you easily find leaf nodes, the size of the subtrees, and other structural properties. Let's put the query and the "housekeeping" into a single procedure instead:

```
CREATE PROCEDURE DropTree (IN downsized CHAR(10))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
DECLARE drop_member CHAR(10);
DECLARE drop_lft INTEGER;
DECLARE drop_rgt INTEGER;

-- save the dropped subtree data with a singleton SELECT
SELECT member, lft, rgt
```

```
  INTO drop_member, drop_lft, drop_rgt
  FROM OrgChart
 WHERE member = downsized;

-- subtree deletion is easy
DELETE FROM OrgChart
 WHERE lft BETWEEN drop_lft and drop_rgt;

-- close up the gap left by the subtree
UPDATE OrgChart
  SET lft = CASE
            WHEN lft > drop_lft
            THEN lft - (drop_rgt - drop_lft + 1)
            ELSE lft END,
      rgt = CASE
            WHEN rgt > drop_lft
            THEN rgt - (drop_rgt - drop_lft + 1)
            ELSE rgt END
 WHERE lft > drop_lft
      OR rgt > drop_lft;
END;
```

A complete procedure should have some error handling, but I am leaving that topic as an exercise for the reader. The expression (drop_rgt - drop_lft + 1) is the size of the gap, and we renumber every node to the right of the gap by that amount. The WHERE clause makes the two ELSE clauses redundant, but they make me feel safer, so I write them anyway.

If you used only the original DELETE FROM statement instead of the procedure just given, or if you build a table from several different sources, you could get multiple gaps that you wish to close. This requires a complete renumbering.

```
UPDATE OrgChart
  SET lft = (SELECT COUNT(*)
               FROM (SELECT lft FROM OrgChart
                       UNION ALL
                      SELECT rgt FROM OrgChart) AS LftRgt (seq)
              WHERE seq <= lft),
      rgt = (SELECT COUNT(*)
               FROM (SELECT lft FROM OrgChart
                       UNION ALL
```

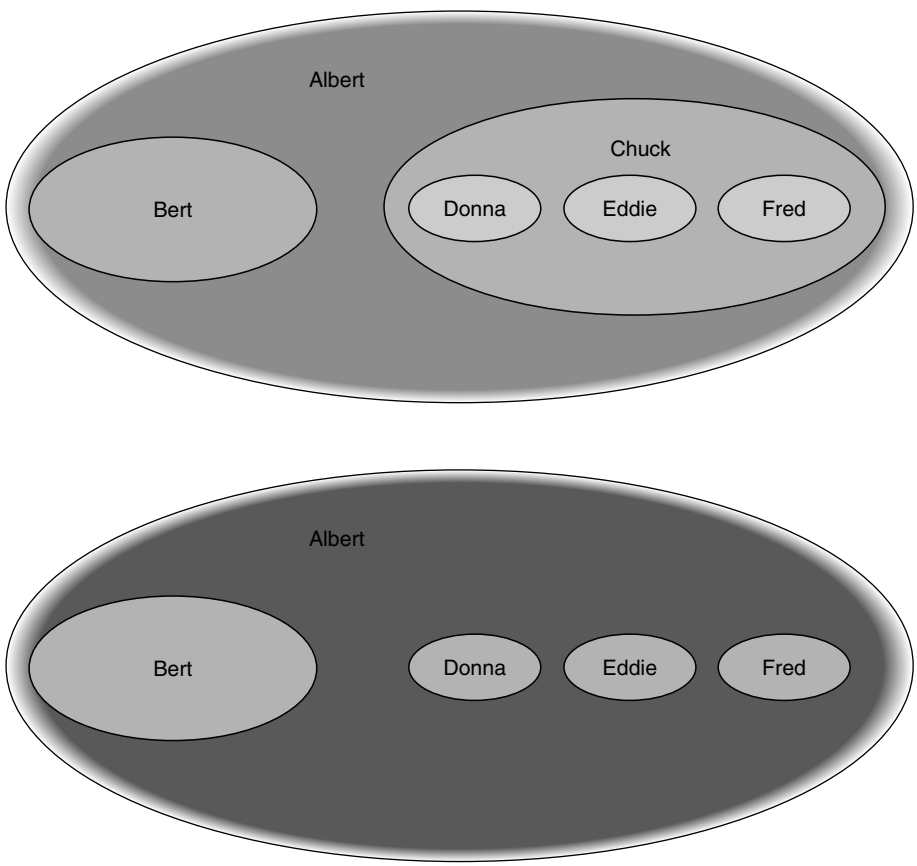
```
                        SELECT rgt FROM OrgChart) AS LftRgt (seq)
WHERE seq <= rgt);
```

If the derived table LftRgt is a bit slow, you can use a temporary table and index it or use a VIEW that will be materialized.

4.5.2 Deleting a Single Node

Deleting a single node in the middle of the tree is harder than removing whole subtrees. When you remove a node in the middle of the tree, you have to decide how to fill the hole. One approach is to put a "vacant position" marker in the organizational chart, so that the structure does not change. This might

Fig. 4.5



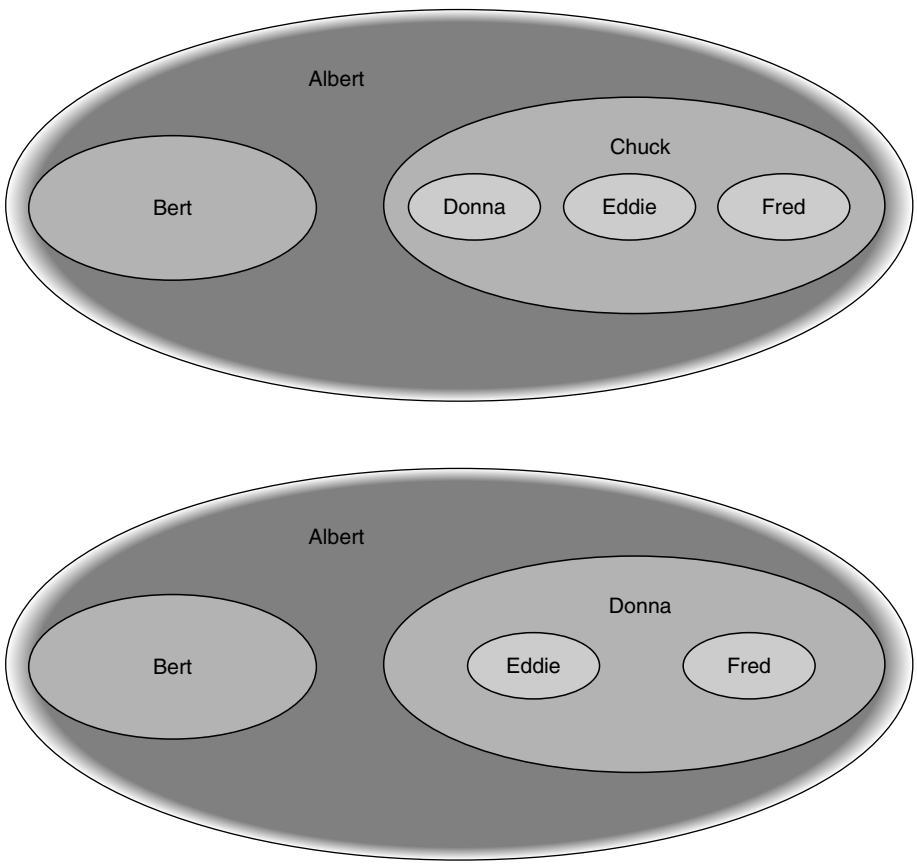


Fig. 4.6

be followed by moving existing personnel into the vacancies as they are created.

There are two basic ways to change the structure when a node is removed. One method is to connect the children to the parent of the original node—Mom dies and Grandma adopts the kids, as shown in Figure 4.5. This happens automatically in the nested set model; you just delete the node and its children are already contained in their ancestor nodes. Now you need to renumber the nodes to the left of the deletion.

The second method is to promote one of the children to the original node's position—Dad dies and the oldest son takes over the business, as shown in Figure 4.6. The oldest child is always shown as the leftmost child node under its parent. There is a problem with this operation, however. If the older child

has children of his own, you have to decide how to handle them and so on down the tree until you get to a leaf node.

Let's use a '{vacant}' as a marker for the vacancy. This way we can promote the oldest subordinate to the vacant job, then decide if we want to fill his previous position with the oldest subordinate.

```
CREATE PROCEDURE Downsize(IN downsized_guy CHAR(10))
LANGUAGE SQL
DETERMINISTIC
UPDATE OrgChart
    SET member
        = CASE WHEN OrgChart.member = downsized_guy
                    AND OrgChart.lft + 1 = OrgChart.rgt -- leaf
                    node
               THEN '{vacant}'
               WHEN OrgChart.member = downsized_guy
                    AND OrgChart.lft + 1 <> OrgChart.rgt --
promote subordinate
               THEN (SELECT O1.member
                        FROM OrgChart AS O1
                       WHERE OrgChart.lft + 1 = O1.lft)
               WHEN OrgChart.member -- vacate subordinate position
                    = (SELECT O1.member
                    FROM OrgChart AS O1
                    WHERE OrgChart.lft + 1 = O1.lft)
               THEN '{vacant}'
               ELSE member END;
```

This leads to the following cases:

1. A leaf node has no subordinates to promote, so leave the node vacant.
2. If there is a subordinate, then we have two steps: promote the subordinate and vacate the subordinate's current position.

4.5.3 Pruning a Set of Nodes from a Tree

An interesting version of this problem is displaying the tree with some of the subtrees pruned from the tree. This is usually a dynamic process that is used for displaying the tree structure in the front end. The most common example is

clicking on the "+" and "-" boxes of a Windows directory display to open and close nested files.

First, build a table for the root nodes of the subtrees you wish to hide:

```
CREATE TABLE Cuts (node CHAR(5) NOT NULL PRIMARY KEY);
```

Next, use a VIEW to drop the subtrees rooted at the cut nodes:

```
CREATE VIEW PrunedTree (node, lft, rgt)
AS SELECT T1.T1.part, T1.lft, T1.rgt
     FROM Tree AS T1, Tree AS T2, Cuts AS C1
    WHERE T1.lft
          NOT BETWEEN T2.lft + 1 AND T2.rgt - 1
      AND C1.part = T2.part
    GROUP BY T1.part, T1.lft, T1.rgt
HAVING COUNT(*) = (SELECT COUNT(*) FROM Cuts);
```

These actions will not renumber the (lft, rgt) pairs, but we can do that if you need it. Otherwise, the BETWEEN predicates for nesting are still valid and are all that is required for displaying the tree.

4.6 Closing Gaps in the Tree

The important thing is to preserve the nested subsets based on (lft, rgt) numbers. As you remove nodes from a tree you create gaps in the nested sets numbers. These gaps do not destroy the subset property, but they can present other problems and should be closed. This is like garbage collection in other languages. The easiest way to understand the code is to break it up into a series of meaningful VIEWs, then use the VIEWs to UPDATE the tree table. This VIEW "flattens out" the whole tree into a list of nested sets numbers, regardless of whether they are lft or rgt numbers.

Let's start with a table of assemblies and add some constraints to it.

```
CREATE TABLE Assemblies
(part CHAR(2) PRIMARY KEY,
lft INTEGER NOT NULL UNIQUE,
rgt INTEGER NOT NULL UNIQUE,
CONSTRAINT valid_lft CHECK (lft > 0),
CONSTRAINT valid_rgt CHECK (rgt > 1),
CONSTRAINT valid_range_pair CHECK (lft < rgt));
```

```
INSERT INTO Assemblies
VALUES ('A', 1, 28);
       ('B', 2, 5);
       ('C', 6, 19);
       ('D', 20, 27);
       ('E', 3, 4);
       ('F', 7, 16);
       ('G', 17, 18);
       ('H', 21, 26);
       ('I', 8, 9);
       ('J', 10, 15);
       ('K', 22, 23);
       ('L', 24, 25);
       ('M', 11, 12);
       ('N', 13, 14);
```

First, we can use a view with all the (lft, rgt) numbers in a single column.

```
CREATE VIEW LftRgt (visit)
AS SELECT lft FROM Assemblies
   UNION
   SELECT rgt FROM Assemblies;
```

This VIEW finds the left numbers in gaps instead of in the tree.

```
CREATE VIEW Firstvisit (visit)
AS SELECT (visit + 1)
     FROM LftRgt
    WHERE (visit + 1) NOT IN (SELECT visit FROM LftRgt)
      AND (visit + 1) > 0;
```

The final predicate is to keep you from going past the leftmost limit of the root node, which is always 1. Likewise, this VIEW finds the right nested sets numbers in gaps instead of in the tree.

```
CREATE VIEW LastVisit (visit)
AS SELECT (visit - 1)
     FROM LftRgt
    WHERE (visit - 1) NOT IN (SELECT visit FROM LftRgt)
      AND (visit - 1) < 2 * (SELECT COUNT(*) FROM LftRgt);
```

The final predicate is to keep you from going past the rightmost limit of the root node, which is twice the number of nodes in the tree. You then use these two VIEWs to build a table of the gaps that have to be closed.

```
CREATE VIEW Gaps (commence, finish, spread)
AS SELECT A1.visit, L1.visit, ((L1.visit - A1.visit) + 1)
     FROM Firstvisit AS A1, LastVisit AS L1
    WHERE L1.visit = (SELECT MIN(L2.visit)
                        FROM LastVisit AS L2
                       WHERE A1.visit <= L2.visit);

CREATE PROCEDURE X1()
LANGUAGE SQL
DETERMINISTIC
WHILE EXISTS (SELECT * FROM Gaps)
   DO UPDATE Assemblies
         SET rgt = CASE
                   WHEN rgt > (SELECT MIN(commence) FROM Gaps)
                   THEN rgt - 1 ELSE rgt END,
             lft = CASE
                   WHEN lft > (SELECT MIN(commence) FROM Gaps)
                   THEN lft - 1 ELSE lft END;
END WHILE;

CREATE VIEW Gaps (commence, finish, spread)
AS SELECT A1.visit, L1.visit, ((L1.visit - A1.visit) + 1)
     FROM Firstvisit AS A1, LastVisit AS L1
    WHERE L1.visit = (SELECT MIN(L2.visit)
                        FROM LastVisit AS L2
                       WHERE A1.visit <= L2.visit);
```

This query will tell you the start and finish nested sets numbers of the gaps, as well as their spread. It makes a handy report in itself, which is why I have shown it with the redundant finish and spread columns. That is not why we created it. It can be used to "slide" everything over to the left, thus:

```
CREATE PROCEDURE X2()
LANGUAGE SQL
DETERMINISTIC
-- This will have to be repeated until gaps disappear
```

```
WHILE EXISTS (SELECT * FROM Gaps)
DO UPDATE Assemblies
   SET rgt = CASE
             WHEN rgt > (SELECT MIN(commence) FROM Gaps)
             THEN rgt - 1 ELSE rgt END,
       lft = CASE
             WHEN lft > (SELECT MIN(commence) FROM Gaps)
             THEN lft - 1 ELSE lft END;
END WHILE;
```

The actual number of iterations is given by comparing the size of the original table and the final size after the gaps are closed. This method keeps the code simple at this level, but the VIEWs under it are tricky and could take a lot of execution time. It would seem reasonable to use the gap size to speed up the closure process, but that can get tricky when more than one node has been dropped.

4.7 Summary Functions on Trees

There are tree queries that deal strictly with the nodes themselves and have nothing to do with the tree structure at all. For example, what is the name of the president of the company? How many people are in the company? Are there two people with the same name working here? These queries are handled with the usual SQL queries, and there are no surprises.

Other types of queries do depend on the tree structure. For example, what is the total weight of a finished assembly (i.e., the total of all of its subassembly weights)? Do Harry and John report to the same boss?

The use of the BETWEEN predicate with a GROUP BY and aggregate functions allows us do to basic hierarchical summaries, such as finding the total salaries of the subordinates of each employee.

```
SELECT O2.member, SUM(O1.salary) AS total_salary_budget
  FROM OrgChart AS O1, Personnel AS O2
 WHERE O1.lft BETWEEN O2.lft AND O2.rgt
 GROUP BY O2.member;
```

Any other aggregate function such as MIN(), MAX(), AVG(), and COUNT() can be used along with CASE expressions and function calls. You can be creative here, but there is one serious problem to watch out for. This query format assumes that the structure within the subtree rooted at each node does not matter.

4.7.1 Iterative Parts Update

Let's consider a sample database that shows a parts explosion for a Frammis in a nested sets representation. A Frammis is the imaginary device that holds those widgets MBA students are always marketing in their textbooks. This is built from the assemblies table we have been using, with extra columns for the quantity and weights of the various assemblies. As an aside, constraint names in SQL-92 must be unique at the schema level, not the table level.

```
CREATE TABLE Frammis
(part CHAR(2) PRIMARY KEY,
qty INTEGER NOT NULL
     CONSTRAINT positive_qty CHECK (qty > 0),
wgt INTEGER NOT NULL
     CONSTRAINT non_negative_wgt CHECK (wgt >= 0),
lft INTEGER NOT NULL UNIQUE
     CONSTRAINT valid_lft CHECK (lft > 0),
rgt INTEGER NOT NULL UNIQUE
     CONSTRAINT valid_rgt CHECK (rgt > 1),
CONSTRAINT valid_range_pair CHECK (lft < rgt));
```

We initially load it with this data:

Frammis

part	qty	wgt	lft	rgt
'A'	1	0	1	28
'B'	1	0	2	5
'C'	2	0	6	19
'D'	2	0	20	27
'E'	2	12	3	4
'F'	5	0	7	16
'G'	2	6	17	18
'H'	3	0	21	26
'I'	4	8	8	9
'J'	1	0	10	15
'K'	5	3	22	23
'L'	1	4	24	25
'M'	2	7	11	12
'N'	3	2	13	14

The leaf nodes are the most basic parts, the root node is the final assembly, and the nodes in between are subassemblies. Each part or assembly has a unique catalog number (in this case one or two letters), a weight, and the quantity of this unit that is required to make the next unit above it.

The Frammis table is a convenient fictional device to keep examples simple. In a real schema for a parts explosion there should be other tables. One such table would be an assembly table to describe the structural relationship of the assemblies. Another would be an inventory or parts table to describe each indivisible part of the assemblies. In addition, there would be tables for suppliers, for estimated assembly times, and so forth. For example, the parts data in the Frammis table might be split out and put into a table, as in this example:

```
CREATE TABLE Parts
(part_id CHAR(2) NOT NULL PRIMARY KEY,
part_name VARCHAR(15) NOT NULL,
wgt INTEGER NOT NULL
    CHECK (wgt >= 0),
supplier_nbr INTEGER NOT NULL
          REFERENCES Suppliers (supplier_nbr),
 ..);
```

The quantity has no meaning in the parts table. If a part is an undividable piece of raw material, it will have a weight and other physical attributes. Thus we might have a wheel made from steel that we buy from an outside supplier that we later replace with a wheel made from aluminum that we buy from a different supplier and substitute into the assemblies that use wheels. It is a different wheel, but it has the same function and quantity as the old wheel.

Likewise, we might stop making our own motors and start buying them from a supplier. The motor assembly would still be in the tree, and it would still be referred to by an assembly code; however, its subordinates would disappear. In effect the "blueprint" for the assemblies is shown in the nesting of the nodes of the assemblies table with quantities added.

The iterative procedure for calculating the weight of any part is fairly straightforward. If the part has no children, just use its own weight. For each of its children, if they have no children, then their contribution is their weight times their quantity. If they do have children, their contribution is the total of the quantity times the weight of all the children.

```
CREATE PROCEDURE WgtCalc_1 ()
LANGUAGE SQL
```

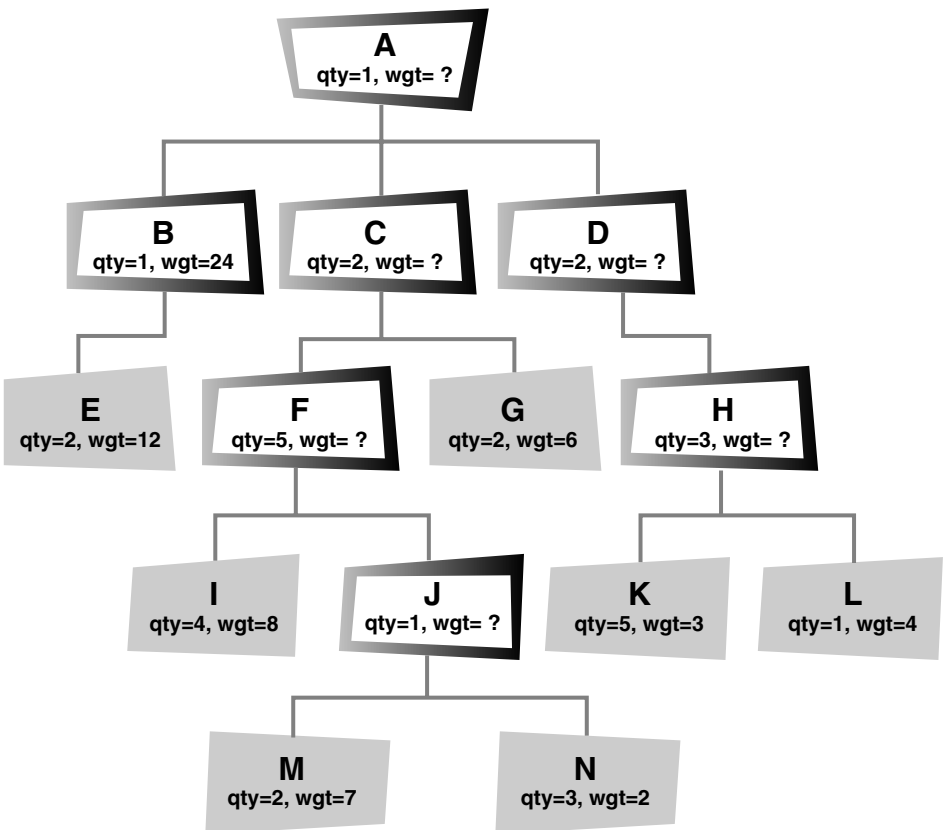
```
DETERMINISTIC
BEGIN
UPDATE Frammis -- clear out the weights
   SET wgt = 0
 WHERE lft < (rgt - 1);
WHILE EXISTS (SELECT * FROM Frammis WHERE wgt = 0)
DO UPDATE Frammis
     SET wgt =
           CASE -- all the children have a weight computed
           WHEN 0 < ALL (SELECT C.wgt
                           FROM Frammis AS C
                                LEFT OUTER JOIN
                                Frammis AS B
                                 ON B.lft
                                    = (SELECT MAX(S.lft)
                                         FROM Frammis AS S
                                        WHERE C.lft > S.lft
                                          AND C.lft < S.rgt)
           WHERE B.part = Frammis.part)
       THEN (SELECT COALESCE (SUM(C.wgt * C.qty), Frammis.wgt)
                       FROM Frammis AS C
                            LEFT OUTER JOIN
                            Frammis AS B
                             ON B.lft
                                = (SELECT MAX(S.lft)
                                     FROM Frammis AS S
                                    WHERE C.lft > S.lft
                                      AND C.lft < S.rgt)
           WHERE B.part = Frammis.part)
          ELSE Frammis.wgt END;
END WHILE;
END;
```

This will give us the following result after moving up the tree one level at a time, as shown in Figures 4.7 through 4.11.

Frammis

part	qty	wgt	lft	rgt
A	1	682	1	28
B	1	24	2	5
C	2	272	6	19

Fig. 4.7



Iteration one, leaf nodes only

(cont.)

part	qty	wgt	lft	rgt
D	2	57	20	27
E	2	12	3	4
F	5	52	7	16
G	2	6	17	18
H	3	19	21	26
I	4	8	8	9
J	1	20	10	15
K	5	3	22	23
L	1	4	24	25
M	2	7	11	12
N	3	2	13	14

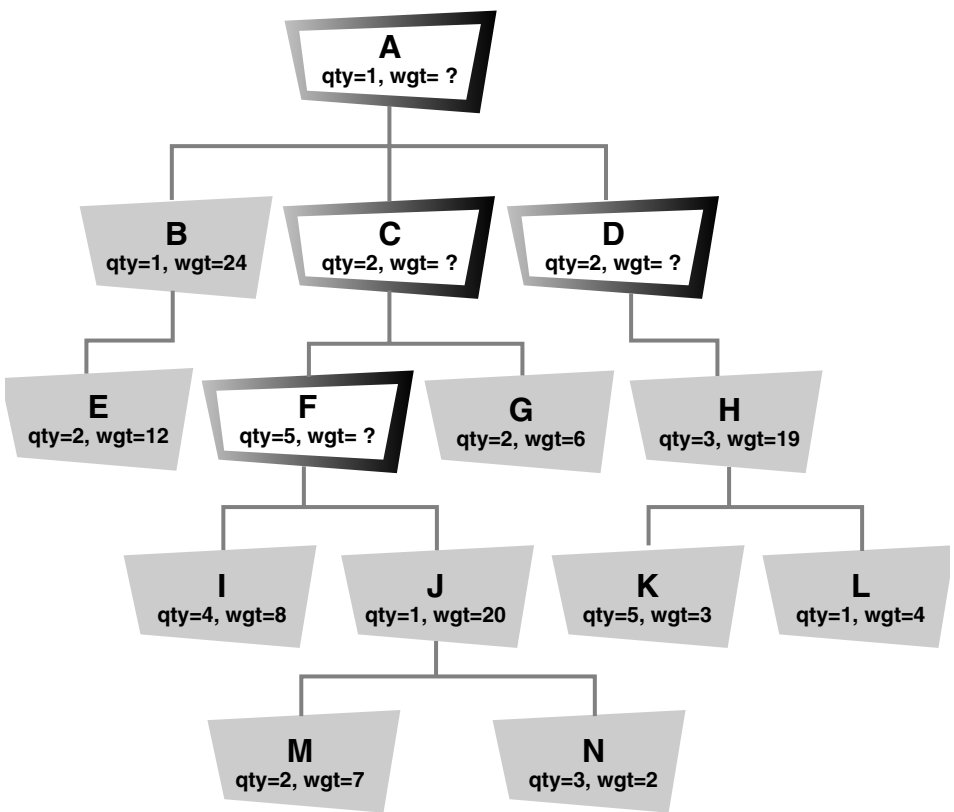


Fig. 4.8

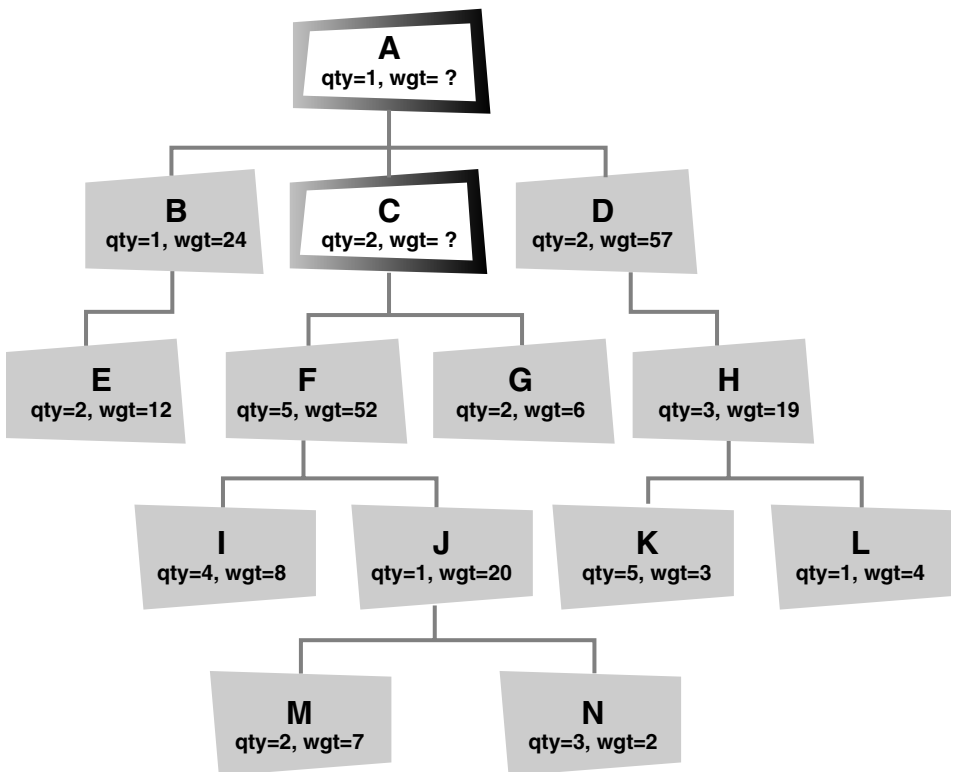
Iteration two

The weight of an assembly will be calculated as the total weight of all its subassemblies. Look at the M and N leaf nodes; the table says that we need two M units weighing 7 kg each, plus three N units weighing 2 kg each, to make one J assembly. Therefore a J assembly weighs ((2 * 7) + (3 * 2)) = 20 kg. This process is iterated from the leaf nodes up the tree, one level at a time until the total weight appears in the root node.

4.7.2 Recursive Parts Update

Let's define a recursive function WgtCalc() that takes a part as an input and returns the weight of that part. To compute the weight the function assumes that the input is a parent node in the tree and sums the quantity times the weight for all the children.

Fig. 4.9



Iteration two

If there are no children, it returns just the parent's weight, which means the node was a leaf node. If any child is itself a parent, the function calls itself recursively to resolve that part's weight.

```
CREATE FUNCTION WgtCalc2 (IN my_part CHAR(2))
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
-- recursive function
RETURN
(SELECT COALESCE(SUM(Subassemblies.qty
           * CASE WHEN Subassemblies.lft + 1 = Subassemblies.rgt
                  THEN Subassemblies.wgt
                  ELSE WgtCalc (Subassemblies.part)
                  END), MAX(Assemblies.wgt))
```

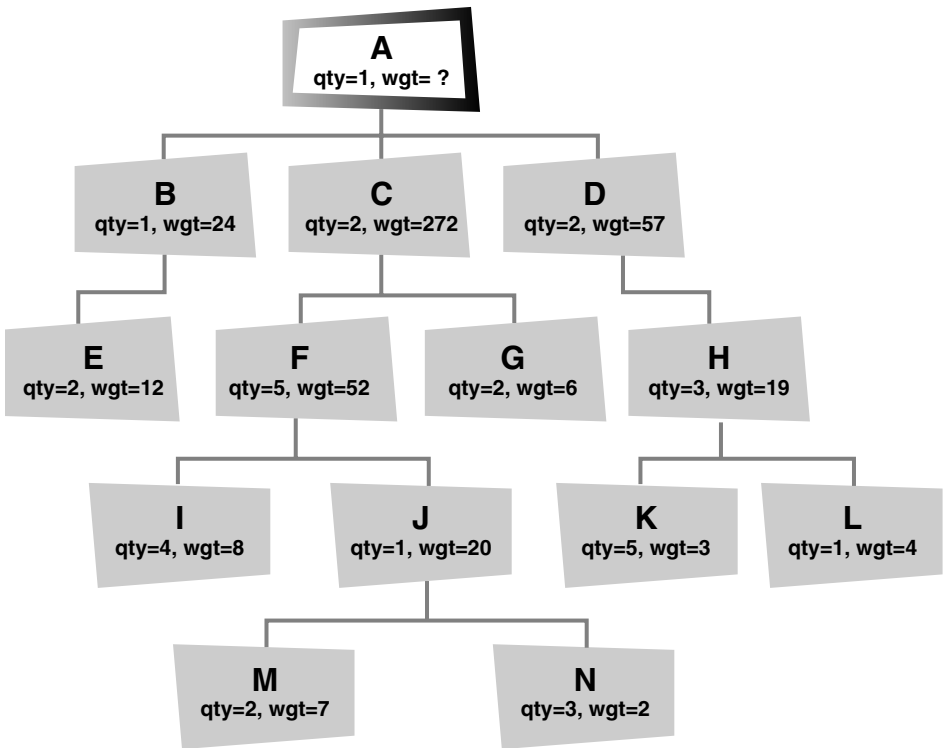


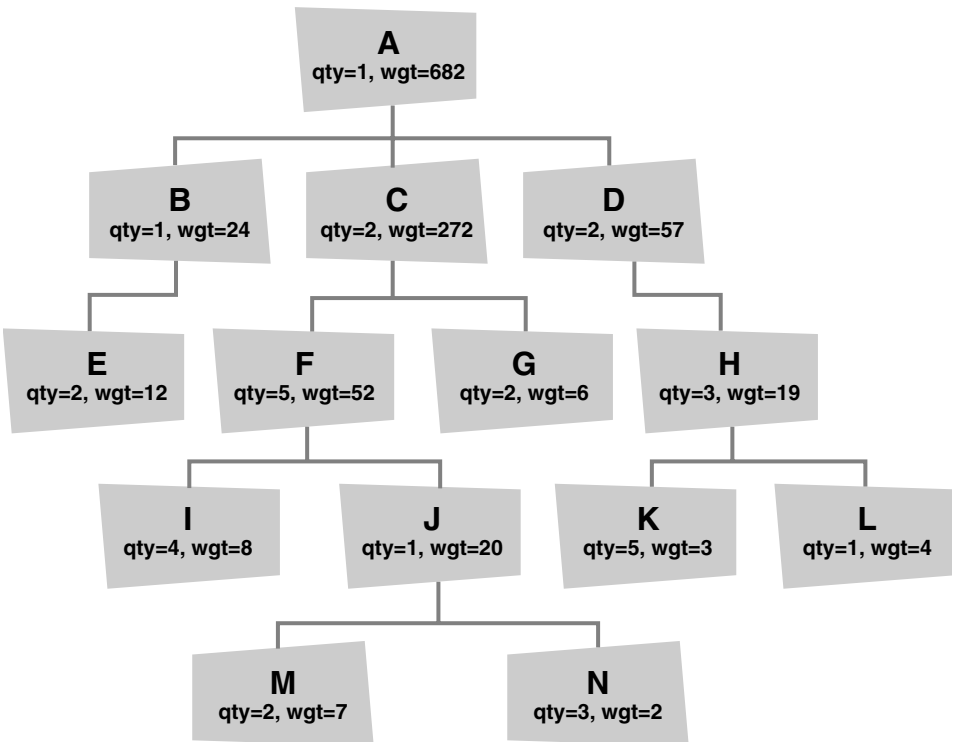
Fig. 4.10

Iteration four

```
FROM Frammis AS Assemblies
     LEFT OUTER JOIN
     Frammis AS Subassemblies
     ON Assemblies.lft < Subassemblies.lft
        AND Assemblies.rgt > Subassemblies.rgt
        AND NOT EXISTS
            (SELECT *
               FROM Frammis
              WHERE lft < Subassemblies.lft
                AND lft > Assemblies.lft
                AND rgt > Subassemblies.rgt
                AND rgt < Assemblies.rgt)
WHERE Assemblies.part = my_part);
```

We can use the function in a VIEW to get the total weight.

Fig. 4.11



Iteration five, the root

```
CREATE VIEW TotalWeight (part, qty, wgt, lft, rgt)
AS
SELECT part, qty, WgtCalc(part, lft, rgt)
  FROM Frammis;
```

Of course, the UPDATE is now trivial...

```
UPDATE Frammis SET wgt = WgtCalc(part);
```

4.8 Inserting and Updating Trees

Updates to the nodes are performed by searching for the key of each node; there is nothing special about them. However, rearranging the structure of the tree is tricky because figuring out the (lft, rgt) nested sets numbers requires a good bit of algebra in a large tree. As a programming project you might want to build a tool that takes a "boxes-and-arrows" graphic and converts it into a series of UPDATE and INSERT statements. Inserting a subtree or a new node involves finding a place in the tree for the new nodes, spreading the other

nodes apart by incrementing their nested sets numbers, and then renumbering the subtree to fit into the gap created. This is the deletion procedure in reverse. First, determine the parent for the node, and then spread the nested sets numbers out two positions to the right.

```
CREATE PROCEDURE InsertNewNode
(IN new_part CHAR(2), IN parent_part CHAR(2),
IN new_qty INTEGER, IN new_wgt INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
DECLARE parent INTEGER;
SET parent = (SELECT rgt
                FROM Frammis
               WHERE part = parent_part);
UPDATE Frammis
   SET lft = CASE WHEN lft > parent
                  THEN lft + 2
                  ELSE lft END,
       rgt = CASE WHEN rgt >= parent
                  THEN rgt + 2
                  ELSE rgt END
 WHERE rgt >= parent;
INSERT INTO Frammis (part, qty, wgt, lft, rgt)
VALUES (new_part, new_qty, new_wgt, parent, (parent + 1));
END;
```

This code is credited to Mark E. Barney (email: Mark.E.Barneym1.irs.gov). The idea is to spread the (lft, rgt) numbers after the youngest child of the parent, G in this case, over by two to make room for the new addition, G1.This procedure will add the new node to the rightmost child position, which helps to preserve the idea of an age order among the siblings.

A slightly different version of the same code will let you add a sibling to the right of a given sibling.

```
CREATE PROCEDURE InsertNewNode
(IN new_part CHAR(2), IN lft_sibling_part CHAR(2),
IN new_qty INTEGER, IN new_wgt INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
```

```
IF (SELECT lft -- the root has no siblings
      FROM Frammis
    WHERE part = lft_sibling_part) = 1
THEN LEAVE insert_on_lft;
ELSE BEGIN
    DECLARE lft_sibling INTEGER;
    SET lft_sibling
         = (SELECT rgt
              FROM Frammis
              WHERE part = lft_sibling_part);
    UPDATE Frammis
       SET lft = CASE WHEN lft < lft_sibling
                      THEN lft ELSE lft + 2 END,
           rgt = CASE WHEN rgt < lft_sibling
                      THEN rgt ELSE rgt + 2 END
     WHERE rgt > lft_sibling;
    INSERT INTO Frammis
           VALUES (new_part, new_qty, new_wgt, (lft_sibling + 1),
(lft_sibling + 2));
           END;
END IF;
END;
```

The reason for giving both blocks of code is a note from Morgan Kelsey about some problems he found using a nested set model for a multithreaded message board. They were doing strange things with replies to posted messages. For example, one would assume this was correct behavior when there are multiple children:

```
--1 message 1
----2 -reply to 1
----3 -reply to 1
----5 -reply to 3
----4 -reply to 1
```

However, there are boards around doing this:

```
--1 message 1
----4 -reply to 1
----3 -reply to 1
----5 -reply to 3
----2 -reply to 1
```

Here's an example: http://boards.gamers.com/messages/overview.asp?name=scstratboard.

When the tree structure is displayed, you have to go down to the right, but then up to read the new messages. Apparently people had taken the first method (i.e., inserting new guy as the rightmost sibling) as the way to do any insertions and blindly implemented it.

4.8.1 Moving a Subtree within a Tree

Yes, it is possible to move subtrees inside the nested sets model for hierarchies. However, we need to get some preliminary things out of the way first. The nested sets model needs a few auxiliary tables to help it. The first is the view we built in Section 4.6.

```
CREATE VIEW LftRgt (i)
AS SELECT lft FROM Tree
   UNION ALL
   SELECT rgt FROM Tree;
```

This is all (lft, rgt) values in a single column. Because we should have no duplicates, we use a UNION ALL to construct the VIEW. Yes, LftRgt can be written as a derived table inside queries, but there are advantages to using a VIEW. Self-joins are much easier to construct. Code is easier to read. If more than one user needs this table, it can be materialized only once by the SQL engine. The next table is a working table to hold subtrees that we extract from the original tree. This could be declared as a local temporary table.

```
CREATE LOCAL TEMPORARY TABLE WorkingTree
(root CHAR(2) NOT NULL,
 node CHAR(2) NOT NULL,
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL,
 PRIMARY KEY (root, node))
ON COMMIT DELETE ROWS;
```

The root column is going to be the value of the root node of the extracted subtree. This gives us a fast way to find an entire subtree via part of the primary key. Although this is not important for the stored procedure discussed here, it is useful for other operations that involve multiple extracted subtrees.

Let me move right to the commented code. The input parameters are the root node of the subtree being moved and the node that is to become its new parent. In this procedure there is an assumption that new siblings are added on the right side of the existing siblings, in effect ordering them by their age.

```
CREATE PROCEDURE MoveSubtree
          (IN my_root CHAR(2),
           IN new_parent CHAR(2))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
DECLARE right_most_sibling INTEGER;
DECLARE subtree_size INTEGER;

-- Cannot move a subtree under itself
    DECLARE Self_reference CONDITION;
-- No such subtree root node
    DECLARE No_such_subtree CONDITION;
-- No such parent node in the tree
    DECLARE No_such_parent_node CONDITION;

body_of_proc:
BEGIN
IF my_root = new_parent
   OR new_parent
    IN (SELECT T1.node
          FROM Tree AS T1, Tree AS T2
         WHERE T2.node = my_root
           AND T1.lft BETWEEN T2.lft AND T2.rgt)
THEN SIGNAL Self_reference; -- error handler invoked here
     LEAVE body_of_proc; -- or leave the block
END IF;

IF NOT EXISTS
   (SELECT *
      FROM Tree
    WHERE node = my_root)
THEN SIGNAL No_such_subtree; -- error handler invoked here
     LEAVE body_of_proc; -- or leave the block
END IF;

IF NOT EXISTS
   (SELECT *
      FROM Tree
    WHERE node = new_parent)
THEN SIGNAL No_such_parent_node; -- error handler invoked here
```

```
    LEAVE body_of_proc; -- or leave the block
END IF;

-- put subtree into working table
INSERT INTO WorkingTree (root, node, lft, rgt)
SELECT my_root, T1.node,
       T1.lft - (SELECT MIN(lft)
                   FROM Tree
                  WHERE node = my_root),
       T1.rgt - (SELECT MIN(lft)
                   FROM Tree
                  WHERE node = my_root)
 FROM Tree AS T1, Tree AS T2
WHERE T1.lft BETWEEN T2.lft AND T2.rgt
 AND T2.node = my_root;

-- remove the subtree from original tree
DELETE FROM Tree
 WHERE node IN (SELECT node FROM WorkingTree);

-- get the spread and location for inserting working tree into
tree
SET right_most_sibling
    = (SELECT rgt
         FROM Tree
        WHERE node = new_parent);

SET subtree_size = (SELECT (MAX(rgt) + 1) FROM WorkingTree);

-- make a gap in the tree
UPDATE Tree
   SET lft = CASE WHEN lft > right_most_sibling
                  THEN lft + subtree_size
                  ELSE lft END,
       rgt = CASE WHEN rgt >= right_most_sibling
                  THEN rgt + subtree_size
                  ELSE rgt END
WHERE rgt >= right_most_sibling;

-- insert the subtree and renumber its rows
INSERT INTO Tree (node, lft, rgt)
```

```
SELECT node,
       lft + right_most_sibling,
       rgt + right_most_sibling
  FROM WorkingTree;

-- close gaps in tree
UPDATE Tree
   SET lft = (SELECT COUNT(*)
                FROM LftRgt
               WHERE LftRgt.i <= Tree.lft),
       rgt = (SELECT COUNT(*)
                FROM LftRgt
               WHERE LftRgt.i <= Tree.rgt);

-- clean out working tree table
DELETE FROM WorkingTree;
END body_of_proc;


END; -- of MoveSubtree
```

As a minor note the variables right_most_sibling and subtree_size could have been replaced with their scalar subqueries in the UPDATE and INSERT INTO statements that follow their assignments. However, that would make the code much harder to read at the cost of only a slight boost in performance.

The final UPDATE statement is a version of the standard self-join trick used to find the ordinal position of a value in a column.

I also used this code to show how error handling is done in the SQL/PSM Standard language. You can declare error conditions and then use the SIGNAL statement to put their names into the diagnostics area when they are detected by a handler and some action is taken. The LEAVE command voids out the actions of the labeled block of code in which it appears and jumps control to the end of the block. In this sample code LEAVE is never executed because the SIGNAL terminates execution immediately, and a SIGNAL that was caught and handled would determine whether the block's actions are "voided."

This is one of the few times I will show you possible error handling or even the deferring of constraints. Each vendor's procedural language will be different, and you will have to adjust this code to your product in the real world.

4.8.2 MoveSubtree, Second Version

Another version of the MoveSubtree procedure that does not use the WorkingTree table looks like this:

```
CREATE PROCEDURE MoveSubtree
     (IN my_root CHAR(2), IN new_parent CHAR(2))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
DECLARE origin_lft INTEGER;
DECLARE origin_rgt INTEGER;
DECLARE new_parent_rgt INTEGER;

SELECT lft, rgt
  INTO origin_lft, origin_rgt
  FROM Tree
 WHERE node = my_root;

SET new_parent_rgt
    = (SELECT rgt
         FROM Tree
        WHERE node = new_parent);

UPDATE Tree
   SET lft
       = lft
         + CASE
           WHEN new_parent_rgt < origin_lft
           THEN CASE

                WHEN lft BETWEEN origin_lft AND origin_rgt
                THEN new_parent_rgt - origin_lft
                WHEN lft BETWEEN new_parent_rgt
                             AND origin_lft - 1
                THEN origin_rgt - origin_lft + 1
                ELSE 0 END
           WHEN new_parent_rgt > origin_rgt
           THEN CASE
                WHEN lft BETWEEN origin_lft
                             AND origin_rgt
           THEN new_parent_rgt - origin_rgt - 1
```

```
                WHEN lft BETWEEN origin_rgt + 1
                             AND new_parent_rgt - 1
                THEN origin_lft - origin_rgt - 1
                ELSE 0 END
           ELSE 0 END,

rgt
   = rgt
     + CASE
       WHEN new_parent_rgt < origin_lft
       THEN CASE

            WHEN rgt BETWEEN origin_lft
                         AND origin_rgt
       THEN new_parent_rgt - origin_lft
       WHEN rgt BETWEEN new_parent_rgt AND origin_lft - 1
       THEN origin_rgt - origin_lft + 1
       ELSE 0 END
      WHEN new_parent_rgt > origin_rgt
      THEN CASE
            WHEN rgt BETWEEN origin_lft
                         AND origin_rgt
            THEN new_parent_rgt - origin_rgt - 1
            WHEN rgt BETWEEN origin_rgt + 1
                         AND new_parent_rgt - 1
            THEN origin_lft - origin_rgt - 1
            ELSE 0 END
       ELSE 0 END;
END; -- Movesubtree
```

This code is credited to Alejandro Izaguirre. It does not set a warning if the subtree is moved under itself, but leaves the tree unchanged. Again, the calculations for origin_lft, origin_rgt, and new_parent_rgt could be put into the UPDATE statement as scalar subquery expressions, but the code would be more difficult to read.

4.8.3 Subtree Duplication

In many hierarchies, subtrees are repeated in different parts of the structure. The same subassembly might appear under many different assemblies. In the military, squads, platoons, divisions, and so forth are defined by a known

collection of soldiers, each with particular MOS (military occupational skills). It would be nice to be able to copy the structure of a subtree under a different root node.

Consider a simple tree, where we are going to duplicate the node values in each copy of the structure. Obviously, duplicated nodes cannot be keys, so we have to use the (lft, rgt) pairs instead.

```
CREATE TABLE Tree
(node VARCHAR(5) NOT NULL,
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL,
 PRIMARY KEY (lft, rgt));
```

Let's do this problem in steps, with the calculations explained, and then consolidate everything into one procedure.

1. We need to find the rightmost position of the node that will be the new parent of the copy of the subtree.
2. We need to find out how big the subtree is so we can make a gap for it in the new parent's (lft, rgt) range.
3. We need to insert the copy, renumbering the (lft, rgt) pairs to fill the gap we just made. This is like moving a subtree, but the original subtree is neither deleted in the process, nor do we need a working table to hold the subtree.

```
CREATE PROCEDURE CopyTree
       (IN new_parent VARCHAR(5),
        IN subtree_root VARCHAR(5))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
-- create the gap
UPDATE Tree
   SET lft = CASE WHEN lft > (SELECT rgt
                                FROM Tree
                               WHERE node = new_parent)
                  THEN lft + (SELECT (rgt - lft + 1)
                                FROM Tree
                               WHERE node = subtree_root)
                  ELSE lft END,
```

```
        rgt = CASE WHEN rgt >= (SELECT rgt
                                  FROM Tree
                                  WHERE node = new_parent)
                   THEN rgt + (SELECT (rgt - lft + 1)

                                  FROM Tree
                                WHERE node = subtree_root)
                                ELSE rgt END
              WHERE rgt >= (SELECT rgt
                                  FROM Tree
                                WHERE node = new_parent);

-- insert the copy
INSERT INTO Tree (node, lft, rgt)
SELECT T1.node || '2',
       T1.lft
       + (SELECT rgt - lft + 2
            FROM Tree
          WHERE node = subtree_root),
       T1.rgt
       + (SELECT rgt - lft + 2
            FROM Tree
          WHERE node = subtree_root)
 FROM Tree AS T1, Tree AS T2
WHERE T2.node = subtree_root
  AND T1.lft BETWEEN T2.lft AND T2.rgt;

END;
```

I gave the new nodes a name with a digit '2' appended to them; however, that is to make the results easier to read and is not required.

This little renaming trick also solves another problem you must consider. If I try to copy a subtree under itself, I may have a recursive relationship that is infinite or impossible. Consider a parts explosion that has a subassembly 'X' in which one of the components is another 'X', in which this second 'X' in turn has to contain a third 'X' to work, and so forth.

You might want to add the predicate to assure that this does not happen.

```
CONSTRAINT new_parent
NOT BETWEEN (SELECT lft FROM Tree WHERE node = subtree_root)
       AND (SELECT rgt FROM Tree WHERE node = subtree_root)
```

4.8.4 Swapping Siblings

The following solution for swapping the positions of two siblings under the same parent node is credited to Michel Walsh (Vanderghast@msn.com) and originally appeared in a posting on the MS-SQL Server Newsgroup. If the leftmost sibling has (lft, rgt) = (i0, i1), and the other subtree, the rightmost sibling, has (i2, i3), implicitly, we know that (i0 < i1 < i2 < i3).

With a little algebra we can figure out that if (I) is a lft or rgt value in the table between i0 and i3, then:

1. If (i BETWEEN i0 AND i1), then (i) should be updated to (i + i3 - i1).
2. If (i BETWEEN i2 AND i3), then (i) should be updated to (i + i0 - i2).
3. If (i BETWEEN i1 + 1 AND i2 - 1), then (i) should be updated to (i0 + i3 + i - i2 - i1).

All of this becomes a single update statement, but we will put the (lft, rgt) pairs of the two siblings into local variables so a human being can read the code.

```
CREATE PROCEDURE SwapSiblings
  (IN lft_sibling CHAR(2), IN rgt_sibling CHAR(2))
LANGUAGE SQL
DETERMINISTIC

BEGIN ATOMIC
DECLARE i0 INTEGER;
DECLARE i1 INTEGER;
DECLARE i2 INTEGER;
DECLARE i3 INTEGER;
SET i0 = (SELECT lft FROM Tree WHERE node = lft_sibling);
SET i1 = (SELECT rgt FROM Tree WHERE node = lft_sibling);
SET i2 = (SELECT lft FROM Tree WHERE node = rgt_sibling);
SET i3 = (SELECT rgt FROM Tree WHERE node = rgt_sibling);

UPDATE Tree
   SET lft = CASE WHEN lft BETWEEN i0 AND i1
                  THEN i3 + lft - i1

                  WHEN lft BETWEEN i2 AND i3
```

```
                    THEN i0 + lft - i2
                    ELSE i0 + i3 + lft - i1 - i2 END,
         rgt = CASE WHEN rgt BETWEEN i0 AND i1
                    THEN i3 + rgt - i1
                    WHEN rgt BETWEEN i2 AND i3
                    THEN i0 + rgt - i2
                    ELSE i0 + i3 + rgt - i1 - i2 END
WHERE lft BETWEEN i0 AND i3
  AND i0 < i1
  AND i1 < i2
  AND i2 < i3;
END;
```

4.9 Converting Nested Sets Model to Adjacency List

Most SQL databases have used the adjacency list model for two reasons. The first reason is that in the early days of the relational model Dr. E. F. Codd published a paper using the adjacency list, and he was the final authority. The second reason is that the adjacency list is a way of "faking" pointer chains, the traditional programming method in procedural languages for handling trees.

To convert a nested set model into an adjacency list model use the following query:

```
SELECT B.member AS boss, P.member
  FROM OrgChart AS P
     LEFT OUTER JOIN
     Personnel AS B
     ON B.lft = (SELECT MAX(S.lft)
                   FROM OrgChart AS S
                  WHERE P.lft > S.lft
                    AND P.lft < S.rgt);
```

This single statement, originally written by Alejandro Izaguirre, replaces my own previous attempt that was based on a pushdown stack algorithm. Once more we see that the best way to program SQL is to think in terms of sets and not procedures.

Another version of the same query is credited to Ben-Nes Michael of Italy.

```
SELECT B.member AS boss, P.member
 FROM OrgChart AS B, Personnel AS P
```

```
WHERE P.lft BETWEEN B.lft AND B.rgt
 AND B.member
     = (SELECT MAX(S.member)
          FROM OrgChart AS S
         WHERE S.lft < P.lft
           AND S.rgt > P.rgt);
```

Michael found this was faster and simpler, according to the EXPLAIN results in PostgreSQL. However, the Ben-Nes version does not produce a (NULL, <root>) row in the result set, only the edges of the graph.

4.10 Converting Adjacency List to Nested Sets Model

To convert an adjacency list model to a nested sets model use this bit of SQL/PSM code. It is a simple pushdown stack algorithm, and it is shown without any error handling. The first step is to create tables for the adjacency list data and one for the nested sets model.

```
-- Tree holds the adjacency model
CREATE TABLE Tree
(node CHAR(10) NOT NULL,
 parent CHAR(10));
-- Stack starts empty, will holds the nested set model
CREATE TABLE Stack
(stack_top INTEGER NOT NULL,
 node CHAR(10) NOT NULL,
 lft INTEGER,
 rgt INTEGER);
```

The stack table will be used as a pushdown stack and will hold the final results. The extra column "stack_top" holds an integer that tells you what the current top of the stack is.

```
CREATE PROCEDURE AdjToNested()
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC

DECLARE lft_rgt INTEGER;
DECLARE max_lft_rgt INTEGER;
DECLARE current_top INTEGER;
```

```
SET lft_rgt = 2;
SET max_lft_rgt = 2 * (SELECT COUNT(*) FROM Tree);
SET current_top = 1;

-- clear the stack
DELETE FROM Stack;
-- push the root
INSERT INTO Stack
SELECT 1, node, 1, max_lft_rgt
  FROM Tree
 WHERE parent IS NULL;

-- delete rows from tree as they are used
DELETE FROM Tree WHERE parent IS NULL;

WHILE lft_rgt <= max_lft_rgt - 1
   DO IF EXISTS (SELECT *
                   FROM Stack AS S1, Tree AS T1
                  WHERE S1.node = T1.parent
                    AND S1.stack_top = current_top)
      THEN BEGIN -- push when top has subordinates and set lft
value
           INSERT INTO Stack
           SELECT (current_top + 1), MIN(T1.node), lft_rgt, NULL
             FROM Stack AS S1, Tree AS T1
           WHERE S1.node = T1.parent
             AND S1.stack_top = current_top;
-- delete rows from tree as they are used
  DELETE FROM Tree
  WHERE node = (SELECT node
                  FROM Stack
                  WHERE stack_top = current_top + 1);
   -- housekeeping of stack pointers and lft_rgt
   SET lft_rgt = lft_rgt + 1;
   SET current_top = current_top + 1;
   END;
ELSE BEGIN -- pop the stack and set rgt value
     UPDATE Stack
        SET rgt = lft_rgt,
            stack_top = - stack_top -- pops the stack
      WHERE stack_top = current_top;
```

```
      SET lft_rgt = lft_rgt + 1;
      SET current_top = current_top - 1;
      END;
  END IF;
END WHILE;
-- stack top is not needed in final answer

IF EXISTS (SELECT * FROM Tree)
THEN « error handling for orphans in original tree »
END IF;
END;
```

4.11 Separation of Edges and Nodes

One of the most important features of a model for hierarchies is the separation of edges and nodes. The personnel of a company are entities, and the organizational chart for the company is a relationship among those entities. Because they are different kinds of things they need to be in separate tables. Not only is this just good data modeling, but it also has some very practical advantages.

4.11.1 Multiple Structures

As an example, a shoe company had two reporting hierarchies, one for the manufacturing side of the company, which was based on the physical construction of the footwear, and another volatile hierarchy for the marketing department. The marketing hierarchy was based on where and to whom the shoes were sold. For example, steel-toed work boots were one category in the manufacturing reports. However, at that time there were two distinct groups of buyers of steel-toed work boots. One group was made up of construction workers with really big feet, and the other group was teenaged girls who enjoyed punk rock and had really small feet. People with average-sized feet did not wear these boots. For marketing, size was a vital factor, and for manufacturing it was a few switches on a shoe-making machine.

```
CREATE TABLE Shoes
(shoe_nbr INTEGER NOT NULL PRIMARY KEY,
...);
```

```
CREATE TABLE ManufacturingReports
(shoe_nbr INTEGER NOT NULL
          REFERENCES Shoes(shoe_nbr),
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL,
  ...);

CREATE TABLE MarketingReports
(shoe_nbr INTEGER NOT NULL
          REFERENCES Shoes(shoe_nbr),
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL,
  ...);
```

4.11.2 Multiple Nodes

Aaron J. Mackey pointed out that you can attach a variable number of attributes to a node and then make queries based on searching for them. For example, given this general structure:

```
CREATE TABLE Tree
(node INTEGER NOT NULL PRIMARY KEY,
 lft INTEGER NOT NULL UNIQUE,
 rgt INTEGER NOT NULL UNIQUE,
  ...);
```

Now attach various attributes to each node:

```
CREATE TABLE NodeProperty_1

(node INTEGER NOT NULL
      REFERENCES Tree (node)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
value CHAR(15) NOT NULL);

CREATE TABLE NodeProperty_2

(node INTEGER NOT NULL
      REFERENCES Tree (node)
      ON DELETE CASCADE
```

```
      ON UPDATE CASCADE,
value CHAR(15) NOT NULL);
```

Each node may have zero to (n) related properties, each of which has a value. This query gives all the parents of the set defined by nodes that have a particular property.

4.12 Comparing Nodes and Structure

There are really several kinds of equality comparisons when you are dealing with a hierarchy:

1. Same nodes in both tables,
2. Same structure in both tables, without regard to the nodes, and
3. Same nodes in the same positions in the structure in both tables; they are identical.

Let me once more invoke my organizational chart in the nested sets model.

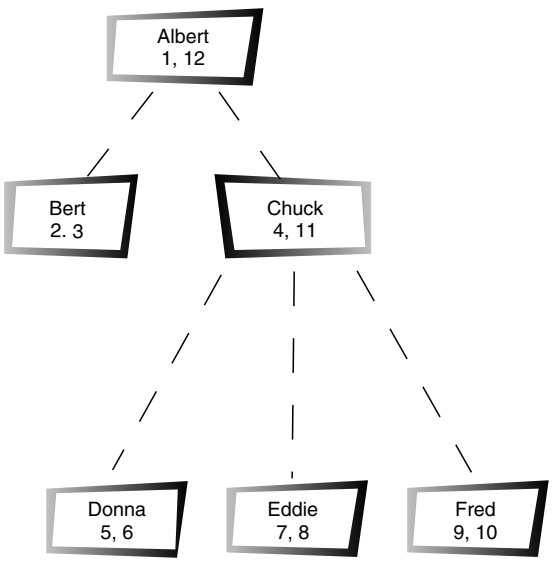
```
CREATE TABLE OrgChart
(member CHAR(10) NOT NULL PRIMARY KEY,
 lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),
 CONSTRAINT order_okay CHECK (lft < rgt));
```

Then let me insert the usual sample data:

OrgChart

member	lft	rgt
'Albert'	1	12
'Bert'	2	3
'Chuck'	4	11
'Donna'	5	6
'Eddie'	7	8
'Fred'	9	10

The organizational chart would look like this as a directed graph:



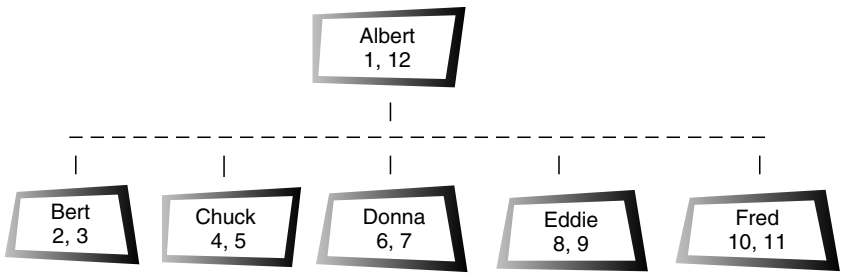
Let's create a second table with the same nodes, but with a different structure:

```
CREATE TABLE OrgChart_2
(member CHAR(10) NOT NULL PRIMARY KEY,
 lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
 rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),
 CONSTRAINT order_okay CHECK (lft < rgt));
```

Insert this table's sample data:

OrgChart_2

member	lft	rgt
'Albert'	1	12
'Bert'	2	3
'Chuck'	4	5
'Donna'	6	7
'Eddie'	8	9
'Fred'	10	11



Now we can do queries based on the set of nodes and on the structure. Let's make a list of variations on such queries.

1. Do we have the same nodes, but in a different structure? One way to do this is with this query:

```
SELECT DISTINCT 'They have different sets of nodes'
 FROM (SELECT * FROM OrgChart
       UNION ALL
       SELECT * FROM OrgChart_2) AS P0 (member, lft, rgt)
 GROUP BY P0.member
HAVING COUNT(*) <> 2;
```

   But do they have to occur the same number of times? That is, if we were to put 'Albert' under 'Donna' in the first organizational chart, how do we count him—once or twice? This is the classic sets versus multisets argument that pops up in SQL all the time. The aforementioned code will reject duplicate multisets. If you want to accept them, use the following code:

```
SELECT DISTINCT 'They have different multi-sets of nodes'
FROM (SELECT DISTINCT *
        FROM OrgChart)
       UNION ALL
     (SELECT DISTINCT *
        FROM OrgChart_2) AS P0 (member, lft, rgt)
 GROUP BY p0.member
HAVING COUNT(*) <> 2;
```

2. Do they have the same structure, but with different nodes? Let's present a table with sample data that has different people inside the same structure as the original personnel table.

OrgChart_3

member	lft	rgt
'Amber'	1	12
'Bobby'	2	3
'Charles'	4	11
'Donald'	5	6
'Edward'	7	8
'Frank'	9	10

The structure is held in the (lft, rgt) pairs, so if they have identical structures, the (lft, rgt) pairs will exactly match each.

```
SELECT DISTINCT 'They have different structures'
FROM (SELECT * FROM OrgChart)
      UNION ALL
     (SELECT * FROM OrgChart_3) AS PO (member, lft, rgt)

GROUP BY PO.lft, PO.rgt

HAVING COUNT(*) <> 2;
```

3. Do they have the same nodes and same structure? In other words, are the trees identical? The logical extension of the other two tests is simply:

```
SELECT DISTINCT 'They are not identical'
FROM (SELECT * FROM OrgChart)
      UNION ALL
     (SELECT * FROM OrgChart_3) AS PO (member, lft, rgt)
 GROUP BY PO.lft, PO.rgt, PO.member
HAVING COUNT(*) <> 2;
```

More often than not you will be comparing subtrees within the same tree. This is best handled by putting the two subtrees into a canonical form. First, you need the root node, and then you can renumber the (lft, rgt) pairs with a derived table of this form:

```
(SELECT O1.member,
      O1.lft - (SELECT MIN(lft)
                  FROM OrgChart
                 WHERE member = :my_member_1) + 1,
      O1.rgt - (SELECT MIN(lft)
```

```
                FROM OrgChart
                WHERE member = :my_member_1) + 1
 FROM OrgChart AS O1, OrgChart AS O2
WHERE O1.lft BETWEEN O2.lft AND O2.rgt
  AND O2.member = :my_member_1) AS P0 (member, lft, rgt);
```

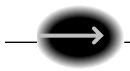
4.13 Nested Sets Code in Other Languages

Flavio Botelho (email: nuncanadaig.com.br) wrote code in MySQL for extracting an adjacency list model from a nested sets model. Although the code depends on the fact that MySQL is not really a relational database, but does sequential processing behind a "near-SQL dialect" language, it is worth passing along. Botelho had seen the outer join query for the conversion (Section 4.9) and wanted to find a faster solution without subqueries, which were not supported in MySQL.

```
SELECT parent_lft = 33; //Change these to fit your needs
SELECT parent_rgt = 102;

SELECT next_brother := parent_lft;

SELECT next_brother :=
       CASE WHEN lft >= next_brother
            THEN rgt + 1
            ELSE next_brother END AS next_brother,
       name, rgt
  FROM Categories
WHERE lft >= parent_lft
   AND rgt <= parent_rgt
HAVING next_brother = rgt + 1
  ORDER BY left;
```

The next_brother stores the right value from the last direct child, so whatever is left comes immediately after this right value and is the next direct child.

So you update the next_brother to this new child, then the HAVING clause will filter to only those children who have the next_brother equal to their right-side sibling. It works in MySQL, but it requires that you are able to change next_brother's value inside the SELECT statement. That is impossible in Standard SQL; you would have to do this with cursors and a loop construct of some kind. Those who like the nested set model and work with MySQL and



PHP may want to look at a PHP library Botelho made to handle nested sets tables in MySQL: http://dev.e-taller.net/dbtree/.

Although it is good to add, update, and delete records, Botelho recommends that you write your own queries to get data from a table instead of using the library function.

There is also a thread or two in the Postgres newsgroups that provides code for manipulating the nested set model. You can start with this link and then explore on your own: http://archives.postgresql.org/pgsql-sql/2002-11/msg00397.php.

For a Java library go to http://www.codebits.com/ntm/java.htm. This library was written by David Medinets, who cautions that you might want to improve it for production work

For ACCESS code go to http://www.mvps.org/access/queries/qry0023.htm.

This page intentionally left blank

Frequent Insertion Trees

THE PROBLEM IN a nested sets tree with frequent insertions is that the left and right (lft, rgt) pairs have to be adjusted so often that locking the table, changing the rows, and unlocking the table again becomes a major overhead. The nested sets model does not require that the union of the rgt and lft numbers be an unbroken sequence to show nesting. All you need are the condition that (lft < rgt), uniqueness of lft and rgt numbers, and that subordination is represented by containment of one (lft, rgt) pair within the ranges of the other (lft, rgt) pairs. This means that we can put gaps into the initial design of the table and fill them without having to reorganize the table each time. The size of the gaps depends on the available physical implementation of exact numeric types and the expected depth of the tree.

The most common example for computer people is the trees in the forest of messages that make up a newsgroup thread (Figure 5.1). A reply to a posting can be inserted anywhere and to almost any depth. The number of messages posting to a newsgroup can also be huge.

As a first attempt at this approach, let's renumber my little organizational chart by multiplying all the lft and rgt numbers by 100.

```
CREATE TABLE OrgChart
(emp CHAR(10) NOT NULL PRIMARY KEY,
lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
rgt INTEGER NOT,
CONSTRAINT order_okay CHECK (lft < rgt));
```

Fig. 5.1

Google
Subject:
Help with Query
Newsgroup:
comp.databases
<< Back | No frame | Sort by date
1 Steve Jul 30, 2000
\-2 Miguel Cruz Jul 31, 2000
|-3 Steve Jul 31, 2000
\-4 Mark E. Rosche Aug 3, 2000
\-5 Art S. Kagel Aug 3, 2000
6 Steve Jan 15, 2004
|-7 wisefool Jan 15, 2004
| \-8 Steve Jan 15, 2004
| |-9 wisefool Jan 15, 2004
| | \-10 Bob Badour Jan 16, 2004
| | |-11 wisefool Jan 16, 2004
| | | \-12 Bob Badour Jan 16, 2004
| | | \-13 wisefool Jan 16, 2004
| | | \-14 Bob Badour Jan 16, 2004
| | | \-15 wisefool Jan 17, 2004
| | \-16 wisefool Jan 16, 2004
| \-17 Lennart Jonsson Jan 16, 2004
| |-18 Bob Badour Jan 16, 2004
| | \-19 Lennart Jonsson Jan 16, 2004
| | \-20 Bob Badour Jan 17, 2004
| | \-21 Lennart Jonsson Jan 17, 2004
| | \-22 Bob Badour Jan 18, 2004
| | \-23 Lennart Jonsson Jan 18, 2004 Jon
| | \-24 Bob Badour Jan 18, 2004
| | \-25 Lennart Jonsson Jan 19, 2004
| | \-26 Bob Badour Jan 19, 2004
| | \-27 Lennart Jonsson Jan 20, 2004
| \-28 Steve Jan 18, 2004
| \-29 Lennart Jonsson Jan 18, 2004
| |-30 Steve Jan 18, 2004
| \-31 Steve Jan 18, 2004
| |-32 Steve Jan 18, 2004
| \-33 Lennart Jonsson Jan 19, 2004
|-34 Bob Badour Jan 15, 2004
|-35 DataMan Jan 16, 2004
\-36 --CELKO-- Jan 16, 2004

OrgChart

emp	lft	rgt
'Albert'	100	1200
'Bert'	200	300
'Chuck'	400	1100
'Donna'	500	600
'Eddie'	700	800
'Fred'	900	1000

The term *spread* will mean the value of (rgt - lft) for one node, and the term *gap* will mean the distance between adjacent siblings under the same parent

node. To insert someone under 'Bert' (e.g., 'Betty'), look at the size of 'Bert's range (300 - 200) and pack the newcomer to the leftmost position, while leaving her node wide enough for more subordinates. One way of doing this is:

```
INSERT INTO OrgChart VALUES ('Betty', 201, 210); -- spread of 9
```

To insert someone under 'Betty,' look at the size of Betty's range (210 - 201) and pack from the left:

```
INSERT INTO OrgChart VALUES ('Bobby', 202, 203); -- spread of 1
```

The new rows should be inserted in the table without locking the table for an update on multiple rows. Assuming you have a 32-bit integer, you can have a depth of nine or ten levels before you have to reorganize the tree. There are two tricks in this approach. First, you must decide on the datatype to use for the (lft, rgt) pairs, and then you must get a formula for the spread size you want to use. Soon you will see that my simple multiplication is not the best way to achieve this goal.

5.1 The Datatype of (lft, rgt)

The (lft, rgt) pairs will obviously be an exact numeric datatype. The goal is to get as wide a numeric range as you can, so that SMALLINT or TINYINT are obviously not going to be considered. The following sections introduce your three choices.

5.1.1 Exploiting the Full Range of Integers

If you don't mind negative numbers, you can use the full range of the integers—something like this on a typical 32-bit machine:

```
INSERT INTO Tree VALUES ('root', -4294967295, 4294967296);
```

I am obviously skipping some of the algebra for computing the spread size, but you get the basic idea. There are some other tricks that involve powers of two and binary trees, but that is another topic. Likewise, some SQL products have a "long" or "big" integer datatype that can be used.

5.1.2 FLOAT, REAL, or DOUBLE PRECISION Numbers

The floating-point numbers give the illusion that the spread can be almost infinite, while truncation and rounding errors will, de facto, impose their own limit. For example, (1000, 2000) impose a limit of 999 integers.

I strongly recommend that you do not use FLOAT or REAL because they will fail when your tree is very deep, as a result of imprecise math. Double precision numbers have the same problems, but they will not show up as early. This is the worst situation—failure occurs when the database is large, and errors are harder to detect.

There is also the problem that many machines used for database applications do not have floating-point hardware. Floating-point math is seldom used in Cobol or commercial applications on mainframes. This means that the floating-point math has to be done in software, which takes longer.

5.1.3 NUMERIC(p,s) or DECIMAL(p,s) Numbers

The DECIMAL(p,s) datatype gives you a greater range than INTEGER in most database products and does not have the rounding problems of FLOAT. Precisons of more than 30 digits are typical; however, you should consult your particular product.

The bad news is that math on DECIMAL(p,s) numbers is often much slower than on either INTEGER or FLOAT. The reason is that most machines do not have hardware support for this datatype, like they do for INTEGER and FLOAT.

5.2 Computing the Spread to Use

There are a number of ways to compute the size of the spread you want to use when you initialize the tree. In the nested sets model the sibling nodes have an order from left to right under their parent node. Given a parent node ('Parent,' x, z), we can assume that the oldest (leftmost) child is of the form ('child_1,' (x + 1), y), in which (x<(x + 1)<y<z). Likewise, in a fully packed, nested set model we would also know the youngest (rightmost) child is of the form ('child_n,' w, (z - 1)), in which (x<w<(z - 1)<z). When we have to insert a new sibling and there is no room in the right gap under his parent, we want to push the existing siblings to the left and leave a gap on the right for the new sibling (Figures 5.2 and 5.3).

First, let's construct a VIEW that will show us what numbers we have for the spread under each parent node.

```
CREATE VIEW Spreads (emp, commence, finish)
AS
SELECT O1.emp, MAX(O2.rgt), (O1.rgt - 1)
  FROM OrgChart AS O1, OrgChart AS O2
  WHERE O2.lft BETWEEN O1.lft AND O1.rgt
```

```
    AND O1.emp <> O2.emp
  GROUP BY O1.emp, O1.rgt
UNION ALL
SELECT O1.emp, (O1.lft + 1), (O1.rgt - 1)
   FROM OrgChart AS O1
  WHERE NOT EXISTS
      (SELECT *
         FROM OrgChart AS O2
        WHERE O2.lft BETWEEN O1.lft AND O1.rgt
          AND O1.emp <> O2.emp)
```

Fig. 5.2

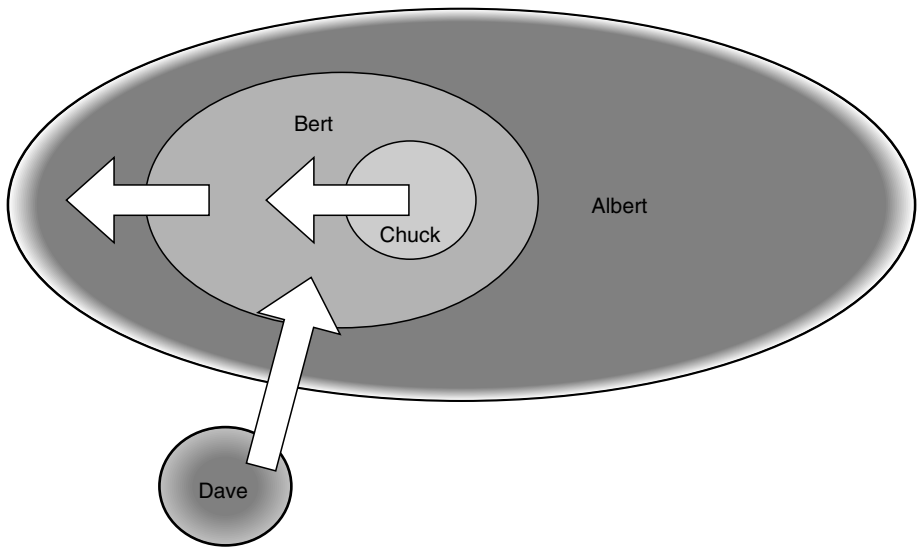
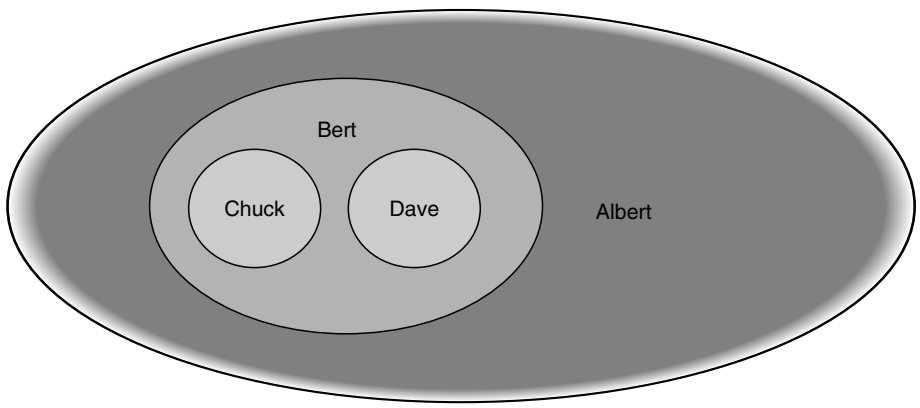


Fig. 5.3



The reason for using a UNION query is that the leaf nodes have no children and will not show up in the SELECT statement of the UNION. We don't need this VIEW, but it makes the code much easier to read than if we folded it into a single statement. The following query shows the real work:

```
CREATE PROCEDURE InsertNewGuy (IN parent CHAR(10), IN new_guy
CHAR(10))
BEGIN ATOMIC
DECLARE commence INTEGER;
DECLARE finish INTEGER;
SET commence = (SELECT commence + 1 FROM Spreads WHERE emp =
parent);
SET finish = (SELECT finish - 1 FROM Spreads WHERE emp =
parent);
IF (finish - commence) <= 0 THEN LEAVE END IF; -- error handling
needed
-- give the new guy 1/10 of the remaining spread
INSERT INTO OrgChart
VALUES (new_guy, commence,
        commence + CAST (((finish - commence)/ 10.0) AS
        INTEGER));
END;
```

What this procedure does is allocate a tenth of the remaining available spread space to each sibling. Perhaps a demonstration will make this easier to see. Using my organizational chart again:

```
DELETE FROM OrgChart;
INSERT INTO OrgChart VALUES ('Albert', 1, 10000000);
```

The maximum depth that a path in this tree can have is 7 because 10^7 = 10,000,000. A different choice of initial width and spread size would give different results. This series of calls will rebuild the original sample tree structure with different (lft, rgt) pairs.

```
CALL InsertNewGuy ('Albert', 'Bert');
CALL InsertNewGuy ('Albert', 'Chuck');
CALL InsertNewGuy ('Chuck', 'Donna');
CALL InsertNewGuy ('Chuck', 'Eddie');
CALL InsertNewGuy ('Chuck', 'Fred');
```

Here are some new rows:

```
CALL InsertNewGuy ('Albert', 'Allen'); -- under the root
CALL InsertNewGuy ('Fred', 'George');
CALL InsertNewGuy ('George', 'Herman');
CALL InsertNewGuy ('Herman', 'Irving');
CALL InsertNewGuy ('Irving', 'Joseph');
CALL InsertNewGuy ('Joseph', 'Kirby'); -- failure!
```

The attempt to insert 'Kirby' fails because the maximum depth is exceeded and the "order_okay" constraint is violated. This is easier to see if we show the spread at each level size as (rgt - lft).

emp	lft	rgt	spread
'Albert'	1	10000000	9999999
'Allen'	1900003	2710002	809999
'Bert'	3	1000002	999999
'Chuck'	1000003	1900002	899999
'Donna'	1000005	1090004	89999
'Eddie'	1090005	1171004	80999
'Fred'	1171005	1243904	72899
'George'	1171007	1178296	7289
'Herman'	1171009	1171737	728
'Irving'	1171011	1171083	72
'Joseph'	1171013	1171019	6

When we insert 'Joseph,' this node only has a range of seven positions, and attempting to divide that range into tenths causes a failure.

We need to consider other ways of determining the divisors and what to do if we need to reorganize the tree because we have nodes where (rgt - lft) = 1 and we wish to insert a new node under them.

5.2.1 Varying the Spread

If you know something about the general shape of the tree (e.g., is it shallow and wide or deep and narrow?), you can replace the constant divisor with either a parameter in the procedure or formula, or with a table lookup subquery expression.

5.2.2 Divisor Parameter

The following query contains a trivial change to the original procedure:

```
CREATE PROCEDURE InsertNewGuy
     (IN parent CHAR(10), IN new_guy CHAR(10), IN divisor
INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE commence INTEGER;
DECLARE finish INTEGER;
DECLARE divisor INTEGER;
SET commence
    = (SELECT commence FROM Spreads WHERE emp = parent);
SET finish
    = (SELECT finish FROM Spreads WHERE emp = parent);
INSERT INTO OrgChart
VALUES (new_guy, commence,
        commence + ((finish - commence)/ divisor));
END;
```

Note that the computation in the last INSERT INTO statement depends on the truncation and rounding rules of your particular product because they are implementation defined in Standard SQL. You might want to use an explicit CAST() expression and, perhaps, truncation and rounding functions. The actual procedure might want to call itself recursively with smaller and smaller spread sizes when it finds a failure caused by an absurdly large spread size. Then if we reach a spread size of one, call a it a reorganization procedure.

5.2.3 Divisor via Formula

The depth of a node in the tree is given by:

```
CREATE VIEW DepthFormula (emp, depth)
AS
SELECT O1.emp, COUNT(O2.emp)
   FROM OrgChart AS O1, OrgChart AS O2
  WHERE O1.lft BETWEEN O2.lft AND O2.rgt
  GROUP BY O1.emp, O1.lft;
```

The root will be at (depth = 1), and the depth will increase as you traverse to the leaf nodes. The depth column in the VIEW can be used as part of a

more complex formula to determine the divisors at each level in the tree. I have shown just the depth itself, but one possible example might be (10^COUNT(*)) or a CASE expression driven by the depth, such as:

```
CASE depth
WHEN 1 THEN 5
WHEN 2 THEN 10
WHEN 3 THEN 25
ELSE 5 END;
```

I do not have any suggestions for the proper formula to use. That would require knowledge of the particular tree's shape.

5.2.4 Divisor via Table Lookup

You can also construct a table of the form:

```
CREATE TABLE DepthDivisors
(depth INTEGER NOT NULL PRIMARY KEY,
divisor INTEGER NOT NULL);
```

or a table of the form:

```
CREATE TABLE EmpDivisors
(emp CHAR(10) NOT NULL PRIMARY KEY,
divisor INTEGER NOT NULL);
```

The first version uses the depth to determine the divisor, so there is an assumption that all the nodes at the same level behave approximately the same in regard to subordinates.

The second version uses the employee to determine the divisor, so there is an assumption that some nodes are expected to have more or fewer subordinates than other nodes.

5.2.5 Partial Reorganization

The simplest reorganization is to return the table to the original nested sets model with a VIEW that gives us all the numbers used in the (lft, rgt) pairs.

```
CREATE VIEW LftRgt (seq)
AS SELECT lft FROM OrgChart
   UNION ALL
   SELECT rgt FROM OrgChart;
```

Then we use that to update the table:

```
UPDATE OrgChart
SET lft = (SELECT COUNT(*)
             FROM LftRgt AS LR
            WHERE LR.seq <= lft),
    rgt = (SELECT COUNT(*)
             FROM LftRgt AS LR
            WHERE LR.seq <= rgt);
```

There is no need for a WHERE clause because all of the nodes will be changed. Unfortunately, we have also destroyed the "big spread" property.

There are several approaches to spreading the (lft, rgt) pairs apart in the usual nested sets model. We can write a query that converts the nested sets model into the adjacency list model, put it into a temporary table, and then pass each (emp, boss) node pair to the InsertNewGuy() procedure, one pair at a time. This is a lot of work, but you get complete control over the reorganization.

The most obvious method is simply to multiply each (lft, rgt) by a constant in the aforementioned UPDATE statement. There are trade-offs in this approach. The code is easy and will close up some of the gaps left by deletions. However, it creates new gaps between siblings. Consider the original OrgChart table with a constant of 100 as the spread we used at the start of this chapter.

OrgChart

emp	lft	rgt
'Albert'	100	1200
'Bert'	200	300
'Chuck'	400	1100
'Donna'	500	600
'Eddie'	700	800
'Fred'	900	1000

We have lost the ranges 1 to 99, 101 to 199, 301 to 399, and so forth to give every node a larger spread of the same proportions. If the insertions are made randomly in the table, it is not a big problem. However, if insertions are made at the leaf nodes or into one particular subtree, then we will be doing this again sooner than if we had planned better.

You will notice that there is a pattern to the gaps we created. The gaps are all of size (spread constant - 1), so we can shift all of the nodes left by that amount, as long as we don't shift a node's (lft, rgt) pair outside the

range of its parent or change the size of the node. This leads us to the next topic.

5.2.6 Rightward Spread Growth

A simpler approach is to increase the spread only to the right of the point where the failure occurred. This can be done by either "stretching" the tree to the right or by "squeezing" some of the nodes to the left at the point of failure. Let's assume that we have captured the node where we failed to insert a new node.

```
UPDATE OrgChart
   SET lft = lft + 100,
       rgt = rgt + 100
WHERE lft > (SELECT rgt
               FROM OrgChart
              WHERE emp = :failure_emp)
   OR rgt >= (SELECT rgt
                FROM OrgChart
               WHERE emp = :failure_emp);
```

The use of a step of 100 is arbitrary and could be replaced by a computation of some sort. The constant is simply easier to code, and I am assuming that the tree will not need reorganization very often.

The other approach is to pack the subordinate nodes to the left to create a larger spread on the right side of the node where the insertion failed.

```
CREATE PROCEDURE ShiftLeft()
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
DECLARE squeeze INTEGER;
SET squeeze
    = (SELECT CASE WHEN MIN(O2.lft - O1.rgt) - 1 > 1
                   THEN MIN(O2.lft - O1.rgt) - 1

                   ELSE 1 END
         FROM OrgChart AS O1, OrgChart AS O2
        WHERE O1.rgt < O2.lft
          AND O1.emp <> O2.emp);
UPDATE OrgChart
```

```
   SET lft = (lft - squeeze),
       rgt = (rgt - squeeze)
WHERE (lft – squeeze) > 0
   AND NOT EXISTS
      (SELECT *
         FROM OrgChart AS O1
        WHERE O1.emp <> OrgChart.emp
          AND (O1.lft
                BETWEEN (OrgChart.lft - squeeze)
          AND (OrgChart.rgt - squeeze)
         OR O1.rgt
         BETWEEN (OrgChart.lft - squeeze)
          AND (OrgChart.rgt - squeeze)));
END;
```

This routine can be executed over and over until all of the children of each node are packed to the left and the largest possible gap is on the right. The problem is that it "slows down" rather quickly and depends on the value of the squeeze parameter.

First call:

emp	lft	rgt
'Albert'	100	1200
'Bert'	101	201
'Chuck'	400	1100
'Donna'	401	501
'Eddie'	601	701
'Fred'	801	901

Second call:

emp	lft	rgt
'Albert'	100	1200
'Bert'	101	201
'Chuck'	400	1100
'Donna'	401	501
'Eddie'	502	602
'Fred'	702	802

Third call:

emp	lft	rgt
'Albert'	100	1200
'Bert'	101	201
'Chuck'	400	1100
'Donna'	401	501
'Eddie'	502	602
'Fred'	701	801

The rightmost node, 'Fred', will continue to shift to the left, but only one step at a time. 'Albert' never gets to (1, 1101), 'Bert' never gets to (2, 102), and so forth.

5.3 Total Reorganization

There may come a time when you need to reorganize the entire table rather than simply shifting part of the table structure. The goal will be to shift all of the nodes over to the left without changing their spread, so as to give the largest possible gap on the right side of the siblings of every parent in the tree. If you need a physical analogy, think of a collection of various sized boxes nested inside each other. Pick up the outermost box and turn it on its left side, so that all the boxes shift to the left

5.3.1 Reorganization with Lookup Table

The following solution is credited to Heinz Huber. Let's start with the original table mentioned at the beginning of this chapter and decide what we want it to look like after reorganization

OrgChart -- reorganized

emp	lft	rgt
'Albert'	1	1101
'Bert'	2	102
'Chuck'	103	803
'Donna'	104	204
'Eddie'	205	305
'Fred'	306	406

The structure and the spreads have remained the same, but the gaps between the employees have been closed by shifting them to the right. This leaves larger gaps on the right side of each row of siblings; for example, 'Fred' has a gap of (803 - 406) = 397 to his right, which means there is room for three more additions to his family, since the spread is 100 at this level.

The problem is that there is no "universal" shift factor. Instead, the shift is different for each employee, based on the gaps at their level in the tree. Let's assume that we don't want to implement a cursor solution. We can add another column to the table to hold the shift factor for each node and fourth column for the preorder traversal order. The problem with a cursor solution is that you need a stack for the rgt column values of all the parents so that you can traverse the tree. This is expensive and not very portable because every product has slightly different cursor implementations.

```
CREATE TABLE OrgChart
(emp CHAR(10) NOT NULL PRIMARY KEY,
lft INTEGER NOT NULL UNIQUE CHECK (lft > 0),
rgt INTEGER NOT NULL UNIQUE CHECK (rgt > 1),
CONSTRAINT order_okay CHECK (lft < rgt),
shift INTEGER, -- null means not yet computed
traversal_nbr INTEGER); -- null means not yet computed
```

Yes, this could all be done with a temporary table, or a second table that joins to the original OrgChart. However, these attributes are part of the tree structure, and having them all in one place makes sense. Let's begin by initializing the table.

```
UPDATE OrgChart
SET shift = NULL,
    traversal_nbr = NULL;
```

The NULLs act as markers for the computations.

```
-- Calculate shift factor within a parent node.
-- Leftmost siblings are computed later.
UPDATE OrgChart
   SET shift
   = lft - 1
     - (SELECT MAX(Siblings.rgt)
          FROM OrgChart AS Siblings
          WHERE Siblings.rgt < OrgChart.lft)
WHERE shift IS NULL
```

```
  AND EXISTS -- has sibling on left side
      (SELECT *
         FROM OrgChart AS Siblings
        WHERE Siblings.rgt < OrgChart.lft);
```

That gives us this result at the leaf nodes:

OrgChart -- step 1

emp	lft	rgt	shift	traversal_nbr
'Albert'	100	1200	NULL	NULL
'Bert'	101	201	NULL	NULL
'Chuck'	400	1100	198	NULL
'Donna'	401	501	199	NULL
'Eddie'	601	701	99	NULL
'Fred'	801	901	99	NULL

Now it is time to look at the parents and shift them and their family:

```
UPDATE OrgChart
SET shift
    = lft - 1
      - (SELECT MAX(Parents.lft)
           FROM OrgChart AS Parents
          WHERE Parents.lft < OrgChart.lft
            AND Parents.rgt > OrgChart.rgt)
WHERE shift IS NULL
   OR (lft - shift)
      < (SELECT MAX(Parents.lft)
           FROM OrgChart AS Parents
          WHERE Parents.lft < OrgChart.lft
            AND Parents.rgt > OrgChart.rgt);
```

OrgChart -- step 2

emp	lft	rgt	shift	traversal_nbr
'Albert'	100	1200	NULL	NULL
'Bert'	101	201	0	NULL
'Chuck'	400	1100	198	NULL
'Donna'	401	501	0	NULL
'Eddie'	601	701	99	NULL
'Fred'	801	901	99	NULL

At this point only the root is still NULL. Shifting it will shift every node in the tree leftward.

```
UPDATE OrgChart
SET shift = lft -- 1
WHERE shift IS NULL;
```

OrgChart -- step 3

emp	lft	rgt	shift	traversal_nbr
'Albert'	100	1200	99	NULL
'Bert'	101	201	0	NULL
'Chuck'	400	1100	198	NULL
'Donna'	401	501	0	NULL
'Eddie'	601	701	99	NULL
'Fred'	801	901	99	NULL

Processing each level of the tree still does not give us the final results. We have not yet applied the shift values. For the shift itself you need another additional column, which contains the preorder traversal sequence.

```
UPDATE OrgChart
SET traversal_nbr
    = (SELECT COUNT(*)
         FROM OrgChart AS Original_OrgChart
        WHERE Original_OrgChart.lft <= OrgChart.lft);
```

OrgChart -- step 4

emp	lft	rgt	shift	traversal_nbr
'Albert'	100	1200	99	1
'Bert'	101	201	0	2
'Chuck'	400	1100	198	3
'Donna'	401	501	0	4
'Eddie'	601	701	99	5
'Fred'	801	901	99	6

Now it is time to do the big shift. Each node is moved leftward by the sum of the gaps to its left, and the order of execution is governed by the preorder traversal.

```
UPDATE OrgChart
SET lft
```

```
    = lft
       - (SELECT SUM(shift)
            FROM OrgChart AS Original_OrgChart
           WHERE Original_OrgChart.traversal_nbr
                 <= OrgChart.traversal_nbr),
  rgt
  = rgt
    - (SELECT SUM(shift)
         FROM OrgChart AS Original_OrgChart
        WHERE Original_OrgChart.traversal_nbr
              <= OrgChart.traversal_nbr);
```

You are now ready to reset the shift and traversal_nbr columns to NULLs. The final answer that results is what we wanted.

```
UPDATE OrgChart
SET shift = NULL,
    traversal_nbr = NULL;
```

OrgChart -- step 5

emp	lft	rgt	shift	traversal_nbr
'Albert'	1	1101	NULL	NULL
'Bert'	2	102	NULL	NULL
'Chuck'	103	803	NULL	NULL
'Donna'	104	204	NULL	NULL
'Eddie'	205	305	NULL	NULL
'Fred'	306	406	NULL	NULL

Hopefully this procedure will not be called very often. It will be expensive to run on a large, deep tree and will probably lock the table while it is running. If you have a tree that is being dynamically altered this much, you might try using the quick but inadequate shift by a constant method first, then call this routine when you can take the application off-line.

5.3.2 Reorganization with Recursion

This solution is credited to Richard Romley. Instead of using a table to hold the shifts, they are computed recursively inside a user-defined function.

```
CREATE FUNCTION LeftShift (IN my_emp CHAR(10))
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
-- recursive
RETURN
  (SELECT CASE WHEN MAX(Par.emp) IS NULL
               THEN 0
               ELSE LeftShift (MAX(Par.emp)) END
          + COALESCE (SUM(Sib.rgt - Sib.lft), 0)
          + COUNT(Sib.emp) + 1
     FROM OrgChart AS E1
          INNER JOIN
          OrgChart AS Par
          ON E1.lft > Par.lft
             AND E1.rgt < Par.rgt
               AND NOT EXISTS
                 (SELECT *
                    FROM OrgChart
                   WHERE lft < E1.lft
                     AND lft > Par.lft
                     AND rgt > E1.rgt
                     AND rgt < Par.rgt)
          LEFT OUTER JOIN
          OrgChart AS Sib
          ON Par.lft < Sib.lft
             AND Par.rgt > Sib.rgt
             AND Sib.lft < E1.lft
   WHERE E1.emp = my_emp);
```

A node can have only zero or one parent(s). The only node without a parent is the root. There can be many siblings to the left of a node, but all the result rows will always have the same value for their parent. The MAX(Par.emp) in the SELECT list returns the value for parent and eliminates the need to do a GROUP BY.

The algorithm says that the new lft value for each employee node equals its parent's new lft value plus the sum of the spreads of all its older siblings (Sib.rgt - Sib.lft + 1), which is the same as SUM(Sib.rgt - Sib.lft) + COUNT(Sib) plus one. Because the spreads will be the same for the new values as they were for the old values, they can be calculated from the old values.

However, the parent's new lft value must be determined—and this is done with a recursive function call.

If a parent exists, the function calls itself to get the parent's new lft value, and this process will continue all the way up the tree until the root is found. The tree navigation takes place via recursive function calls.

5.4 Rational Numbers and Nested Intervals Model

Vadim Tropashko proved that it is possible to use rational numbers. (For those of you who have forgotten your math, these are numbers of the form [a/b], where a and b are both integers.) This would avoid the problems of the floating point rounding errors, but it would require a library of functions for this new datatype. Although nearly every programming language today implements IEEE floating-point numbers, there are some—notably, computational algebra systems such as Maple and Mathematica—that have internal formats for rational or even algebraic and irrational numbers, such as the square root of 2 and e.

Rational numbers and the use of half-open intervals, which are the basis for the temporal model in SQL, are all we would need. Suppose we want to insert a new child of the root node [0, 1) (or if you prefer [0/5, 5/5) to make the math cleaner) between the children bracketed by [1/5, 2/5) and [3/5, 4/5). You can insert new intervals with the gaps on each side. New members can be fit at any position. For example, looking at 4/5 and 5/5, I can fit in a node at [21/25, 23/25) and still have plenty of room for more nodes. Given that my integers in most SQL products can go into the billions, I have a pretty big range of values to use before I would have to reorganize. The algebra for rational numbers is well known. You can find greatest common divisor (GCD) algorithms in any textbook and use them to keep the numerators and denominators as small as possible.

```
CREATE FUNCTION gcd(IN x INTEGER, IN y INTEGER) RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
BEGIN
WHILE x <> y
  DO IF x > y THEN SET x = x - y; END IF;
     IF y > x THEN SET y = y - x; END IF;
END WHILE;
RETURN (x);
END;
```

That query is known as the nested intervals model, and it generalizes nested sets. A child node [clft, crgt] is an (indirect) descendant of a parent node ([plft, prgt] if:

```
((plft <= clft) AND (crgt <= prgt))
```

Adding a child node is never a problem. You use an unoccupied segment [lft_1, rgt_1] within a parent interval [plft, prgt] and insert the new child node at [(2 * lft_1 + rgt_1)/3, (lft_1 + 2 * rgt_1)/3], as shown in Figure 5.4.

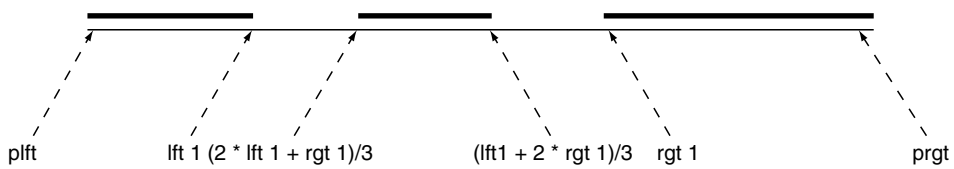
After insertion we still have two more unoccupied segments [lft_1, (2 * lft_1 + rgt_1)/3] and [(rgt_1 + 2 * lft_1)/3, rgt_1] to add more children to the parent node. The problem is that SQL would have to represent the rational (lft, rgt) pairs as pairs of pairs, and the user would have to provide a complete math library for them. If your product supports SQL-99 style user-defined datatypes and functions, this is much easier.

Now we can easily see why nested sets can't model arbitrary directed acyclic graphs; two dimensions are just not enough for representing any partial order.

5.4.1 Partial Order Mappings

Let's introduce a path enumeration model of a tree. You will recognize it as the way that the book you are reading is organized. The path column contains a string of the edges that make up a path from the root ('King') to each node, numbering them from left to right at each level in the tree. This sample organizational chart is from Vadim Tropashko, and we are using it because it is a bit larger and deeper than the examples we have used before; this will help explain the calculations easier.

Fig. 5.4

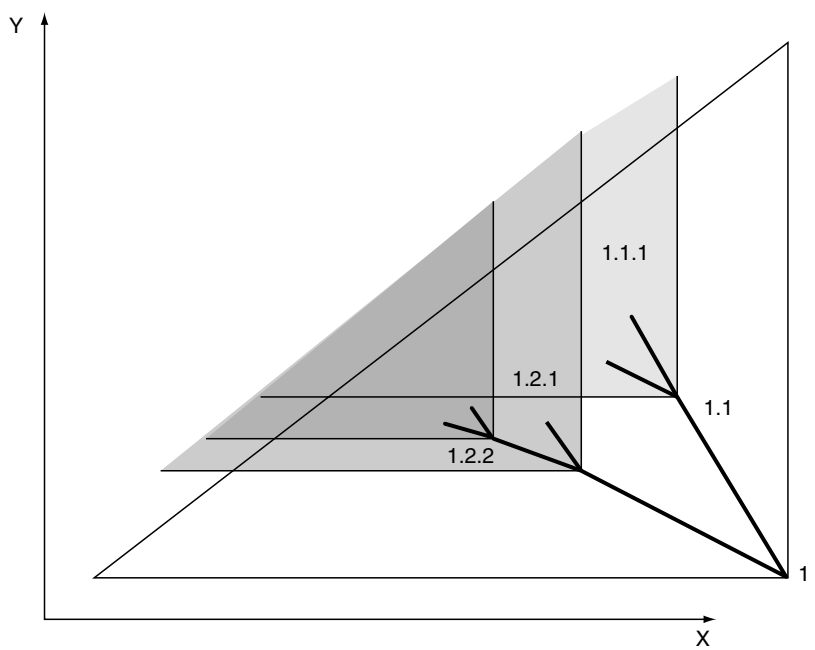


OrgChart

emp_name	path
'King'	'1'
'Jones'	'1.1'
'Scott'	'1.1.1'
'Adams'	'1.1.1.1'
'Ford'	'1.1.2'
'Smith'	'1.1.2.1'
'Blake'	'1.2'
'Allen'	'1.2.1'
'Ward'	'1.2.2'
'Clark'	'1.3'
'Miller'	'1.3.1'

For example, 'Ford' is the second child of the first child ('Jones') of the root ('King'). We are going to turn these directions into numbers shortly, so please be patient.

Fig. 5.5

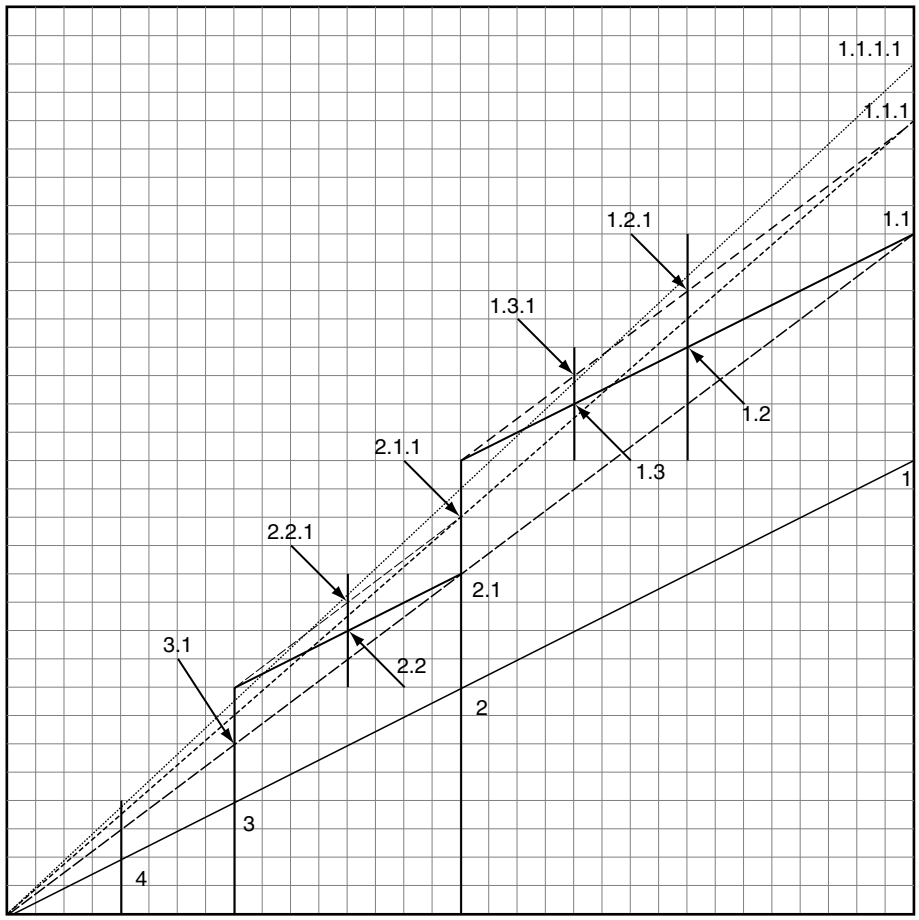


Let's look at the two-dimensional picture of nested intervals and assume that rgt is a horizontal axis x and lft is a vertical axis y. Then the nested intervals tree looks like Figure 5.5.

Each node [lft, rgt] has its descendants bounded within the two-dimensional cone ((y >= lft) AND (x >= rgt)). Because the right interval boundary is always less than the left one, none of the nodes are allowed above the main diagonal, x = y.

The other way to look at this figure is to notice that a child node is a descendant of the parent node whenever a set of all points defined by the child cone ((y >= clft) AND (x <= crgt)) is a subset of the parent cone (y >= plft) AND (x <= prgt). A subset relationship between the cones on the plane is a partial order (Figure 5.6).

Fig. 5.6



We now need to assign pairs of points in the x - y plane that conform to these two constraints.

The choice of a root for the tree is arbitrary, so let's start with the interval [0, 1]. In our geometrical interpretation all the tree nodes belong to the lower triangle of the unit square on the x - y plane. For each node of the tree, let's first define two important points at the x - y plane. The depth-first convergence point is an intersection between the diagonal and vertical lines through the node. For example, the depth-first convergence point for (x = 1, y = 1/2) is (x = 1, y = 1). The breadth-first convergence point is an intersection between the diagonal (x = y) and the horizontal line through the point. For example, the breadth-first convergence point for (x = 1, y = 1/2) is (x = 1/2, y = 1/2). Refer to Figure 5.2 if this is hard to see in your head.

For each parent node, we define the position of the first child as a midpoint halfway between the parent point and depth-first convergence point. You draw a straight line from the parent's point and the depth-first convergence point, and then find the midpoint of that line. Each sibling is defined as a midpoint, halfway between the previous sibling point and breadth-first convergence point. For example, node 2.1 of the OrgChart tree is positioned at the point (x = 1/2, y = 3/8).

Now that the transformation is defined, it is clear which dense domain we are using: not rational or real numbers, but binary fractions. As an aside, the descendant subtree for the parent node "1.2" is a scaled-down replica of the subtree at node "1.1," and the subtree at node "1.1" is a scaled down replica of the tree at node "1," therefore we are a little fractal

5.4.2 Summation of Coordinates

Notice that x and y are not completely independent; we can find both x and y if we know their sum. Therefore we will store two INTEGER numbers—numerator and denominator of the sum of the coordinates x and y—as an encoded node path. Given the numerator and denominator of the rational number that represents the sum of the node coordinates, we can calculate (x, y) coordinates back with this function.

```
CREATE FUNCTION Find_x_numer (IN numer INTEGER, IN denom
INTEGER)
RETURNS INTEGER
BEGIN
DECLARE ret_num INTEGER;
DECLARE ret_den INTEGER;
```

```
   SET ret_num = numer + 1;
   SET ret_den = 2 * denom;
   WHILE FLOOR(ret_num/2) = ret_num/2
      DO SET ret_num = ret_num/2;
         SET ret_den = ret_den/2;
   END WHILE;
   RETURN ret_num;
END;
```

Likewise, there is a function for the denominator of x.

```
CREATE FUNCTION Find_x_denom (IN numer INTEGER, IN denom
INTEGER)
RETURNS INTEGER
BEGIN
DECLARE ret_num INTEGER;
DECLARE ret_den INTEGER;
   SET ret_num = numer + 1;
   SET ret_den = 2 * denom;
   WHILE FLOOR(ret_num/2) = ret_num/2
      DO SET ret_num = ret_num/2;
         SET ret_den = ret_den/2;
   END WHILE;
   RETURN ret_den;
END;
```

The two functions differ from each other by which variable is in the final RETURN statement. Informally, the numer + 1 increment would move the ret_num/ret_den point vertically up to the diagonal. The x coordinate is half of the value, so we just multiplied the denominator by two. Next, we reduced both numerator and denominator by the common power of two.
Naturally, y coordinate is defined as a complement to the sum:

```
CREATE FUNCTION y_numer (IN numer INTEGER, IN denom INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE num INTEGER;
DECLARE den INTEGER;
```

```
   SET num = x_numer(numer, denom);
   SET den = x_denom(numer, denom);
   WHILE den < denom
      DO SET num = 2 * num;
         SET den = 2 * den;
   END WHILE;
   SET num = numer - num;
   WHILE FLOOR(num/2) = num/2
      DO SET num = num/2;
         SET den = den/2;
   END WHILE;
   RETURN num;
END;

CREATE FUNCTION y_denom(IN numer INTEGER, IN denom INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE num INTEGER;
DECLARE den INTEGER;
  SET num = x_numer(numer, denom);
  SET den = x_denom(numer, denom);
  WHILE den < denom
    DO SET num = 2 * num;
       SET den = 2 * den;
  END WHILE;
  SET num = numer - num;
  WHILE FLOOR(num/2) = num/2
    DO SET num = num/2;
       SET den = den/2;
  END WHILE;
  RETURN (den);
END;
```

Now, here's the test (in which 39/32 is the node 1.3.1), using a dummy table:

```
SELECT x_numer(39, 32)||'/'||x_denom(39, 32),
     y_numer(39, 32)||'/'||y_denom(39, 32)
FROM Dummy;
```

Results

5/8	19/32

```
SELECT 5/8 + 19/32, 39/32
  FROM Dummy;
```

Results

1.21875	1.21875

Notice that we did not use floating-point numbers to represent rational numbers and that we wrote all the functions with INTEGER arithmetic instead. In the last test, however, we used a floating point just to verify that 5/8 and 19/32, returned by the previous query, do indeed add up to 39/32.

We'll store two INTEGER numbers—the numerator and denominator of the sum of the coordinates x and y—as an encoded node path. Unlike the pair of integers in the nested sets model, this mapping is stable. The nested intervals model is essentially an enumerated path encoded as a rational number. This is why the OrgChart table was shown as an enumerated path model.

5.4.3 Finding Parent Encoding and Sibling Number

Given the (numer, denom) pair of a child node, we can find the node's parent with the following functions:

```
CREATE FUNCTION parent_numer (IN numer INTEGER, IN denom
INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE ret_num INTEGER;
DECLARE ret_den INTEGER;
   IF numer = 3
   THEN RETURN CAST(NULL AS INTEGER);
   END IF;
   SET ret_num = (numer - 1)/2;
   SET ret_den = denom/2;
   WHILE FLOOR( (ret_num - 1)/4) = (ret_num - 1)/4
      DO SET ret_num = (ret_num + 1)/2;
```

```
          SET ret_den = ret_den/2;
END WHILE;
RETURN ret_num;
END;

CREATE FUNCTION parent_denom (IN numer INTEGER, IN denom
INTEGER)
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE ret_num INTEGER;
DECLARE ret_den INTEGER;
   IF numer = 3
   THEN RETURN CAST(NULL AS INTEGER);
   END IF;
   SET ret_num = (numer - 1)/2;
   SET ret_den = denom/2;
   WHILE FLOOR( (ret_num - 1)/4) = (ret_num - 1)/4
      DO SET ret_num = (ret_num + 1)/2;
         SET ret_den = ret_den/2;
   END WHILE;
   RETURN ret_den;
END;
```

If the node is the root node, it has a numerator of 3 and has no parent. Otherwise, we must move vertically down the x - y plane at a distance equal to the distance from the depth-first convergence point. If the node happens to be the first child, then that is the answer. Otherwise, we must move horizontally at a distance equal to the distance from the breadth-first convergence point until we meet the parent node. Here is the test of the method in which (27/32) is the node '2.1.2' and (7/8) is '2.1':

```
SELECT parent_numer(27, 32)|| '/' ||parent_denom(27, 32)
  FROM Dummy;
```

Results

7/8

In the previous method counting the steps when navigating horizontally would give the sibling number with this function.

```
CREATE FUNCTION sibling_number (IN numer INTEGER, IN denom
INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE ret_num INTEGER;
DECLARE ret_den INTEGER;
DECLARE ret INTEGER;
   IF numer = 3
   THEN RETURN CAST(NULL AS INTEGER);
   END IF;
   SET ret_num = (numer - 1)/2;
   SET ret_den = denom/2;
   SET ret = 1;
   WHILE FLOOR( (ret_num - 1)/4) = (ret_num - 1)/4
       DO IF ret_num = 1
             AND ret_den = 1
          THEN RETURN ret;
          END IF;
   SET ret_num = (ret_num + 1)/2;
   SET ret_den = ret_den/2;
   SET ret = ret + 1;
   END WHILE;
   RETURN ret;
END;
```

The root node is a special stop condition, ret_num = 1 and ret_den = 1, which we can test with:

```
SELECT sibling_number(7, 8) FROM Dummy;
```

Results

```
1
```

5.4.4 Calculating the Enumerated Path and Distance between Nodes

Strictly speaking, we do not have to use an enumerated path, because our encoding is an alternative. On the other hand, an enumerated path provides a

much more intuitive visualization of the node position in the hierarchy, so that we can use the materialized path for input and output of the data if we provide the mapping to our model.

Implementation is a simple application of the methods from the previous sections. We print the sibling number, jump to the parent, and then repeat these two steps until we reach the root:

```
CREATE FUNCTION Path (IN numer INTEGER, IN denom INTEGER)
RETURNS VARCHAR (30)
LANGUAGE SQL
DETERMINISTIC
   IF numer IS NULL
   THEN RETURN ('?');
   ELSE
   RETURN Path(parent_numer(numer, denom),
               parent_denom(numer, denom))
     || '.' || sibling_number(numer, denom);
   END IF;
```

Now we are ready to write a function that takes two nodes, P and C, and tells us when P is the parent of C. A more general query would return the number of levels between P and C, if C is reachable from P and some exception indicator.

```
CREATE FUNCTION Distance (IN num1 INTEGER, IN den1 INTEGER, IN
                    num2 INTEGER, IN den2 INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
RETURN CASE
       WHEN num1 = num2 AND den1 = den2 -- same node
       THEN 0
       WHEN num1 IS NULL -- missing data
       THEN CAST (NULL AS INTEGER)
       ELSE (1 + Distance(parent_numer(num1, den1),
                          parent_denom(num1, den1), num2, den2))
       END;
```

Test the query.

```
SELECT Distance (27, 32, 3, 4) FROM Dummy;
```

Results

2

Negative numbers are interpreted as exceptions. If the (num1/den1) node is not reachable from (num2/den2), the navigation converges to the root. The alternative way to answer whether two nodes are connected is by simply calculating the (x, y) coordinates and checking if the parent interval encloses the child. A more thorough implementation of the method would involve a domain of integers and rational numbers with an unlimited range, like those kinds of numbers that are supported by computer algebra systems, so that a comparison operation would be part of the compiler.

Our system would not be complete without a function inverse to the path, which returns a node's (numer/denom) value once the path is provided. Let's introduce two auxiliary functions, first:

```
CREATE FUNCTION Child_Numerator
(IN num INTEGER, IN den INTEGER, IN child INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
RETURN (num * (child * child) + 3 - (child * child));
```

And likewise, the matching function:

```
CREATE FUNCTION Child_Denominator
(IN num INTEGER, IN den INTEGER, IN child INTEGER)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
RETURN den * (child * child);
```

For example, the third child of the node '1' (encoded as 3/2) is the node '1.3' (encoded as 19/16). The path encoding function is:

```
CREATE FUNCTION Path_Numer(path VARCHAR)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE num INTEGER;
DECLARE den INTEGER;
```

```
DECLARE postfix VARCHAR(1000);
DECLARE sibling VARCHAR(100);
   SET num = 1;
   SET den = 1;
   SET postfix = '.' || path || '.';
   WHILE CHAR_LENGTH(postfix) > 1
      DO SET sibling = SUBSTRING(postfix FROM 2 FOR
INSTR(postfix, '.', 2) - 2);
         SET postfix = SUBSTRING(postfix FROM INSTR(postfix,
'.', 2) FOR CHAR_LENGTH(postfix) - INSTR(postfix, '.', 2) + 1);
         SET num = Child_Numer(num, den, CAST(sibling AS
INTEGER));
         SET den = Child_Denom(num, den, CAST(sibling AS
INTEGER));
END WHILE;
RETURN num;
END;
```

The function INSTR() is a version of the POSITION() function that returns the nth occurrence of the second parameter string within the first parameter string. Again, the corresponding function for the denominator is:

```
CREATE FUNCTION Path_Denom(path VARCHAR)
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE num INTEGER;
DECLARE den INTEGER;
DECLARE postfix VARCHAR(1000);
DECLARE sibling VARCHAR(100);
   SET num = 1;
   SET den = 1;
   SET postfix = '.' || path || '.';
   WHILE CHAR_LENGTH(postfix) > 1
      DO SET sibling = SUBSTRING(postfix FROM 2 FOR
INSTR(postfix, '.', 2) - 2);
         SET postfix = SUBSTRING(postfix FROM INSTR(postfix,
'.', 2) FOR CHAR_LENGTH(postfix) - INSTR(postfix, '.', 2) + 1);
         SET num = Child_Numer(num, den, CAST(sibling AS
INTEGER));
```

```
          SET den = Child_Denom(num, den, CAST(sibling AS
INTEGER));
END WHILE;
RETURN den;
END;

SELECT Path_Numer('2.1.3') || '/' ||
       Path_Denom('2.1.3')
  FROM Dummy;
```

Results

51/64

5.4.5 Building a Hierarchy

Let's create the OrgChart hierarchy in the following table:

```
CREATE TABLE OrgChart
(name VARCHAR(30) NOT NULL UNIQUE,
 numer INTEGER NOT NULL,
 denom INTEGER NOT NULL,
 UNIQUE (numer, denom));

INSERT INTO OrgChart
VALUES ('King', Path_Numer('1'), Path_Denom('1')),
       ('Jones', Path_Numer('1.1'), Path_Denom('1.1')),
       ('Scott', Path_Numer('1.1.1'), Path_Denom('1.1.1')),
       ('Adams', Path_Numer('1.1.1.1'), Path_Denom('1.1.1.1')),
       ('Ford', Path_Numer('1.1.2'), Path_Denom('1.1.2')),
       ('Smith', Path_Numer('1.1.2.1'), Path_Denom('1.1.2.1')),
       ('Blake', Path_Numer('1.2'), Path_Denom('1.2')),
       ('Allen', Path_Numer('1.2.1'), Path_Denom('1.2.1')),
       ('Ward', Path_Numer('1.2.2'), Path_Denom('1.2.2')),
       ('Martin', Path_Numer('1.2.3'), Path_Denom('1.2.3')),
       ('Turner', Path_Numer('1.2.4'), Path_Denom('1.2.4')),
       ('Clark', Path_Numer('1.3'), Path_Denom('1.3')),
       ('Miller', Path_Numer('1.3.1'), Path_Denom('1.3.1'));
```

All the functions written in the previous sections are conveniently combined in a single view:

```
CREATE VIEW Hierarchy (name, numer, denom,
                numer_lft, denom_lft,
                numer_rgt, denom_rgt,
                 path, depth)
AS SELECT name, numer, denom,
          y_numer(numer, denom),
          y_denom(numer, denom),
          x_numer(numer, denom),
          x_denom(numer, denom),
          path (numer, denom),
          Distance(numer, denom, 3, 2)
   FROM OrgChart;
```

Finally, we can create the hierarchical reports.

5.4.6 Depth-first Enumeration by Left Interval Boundary

The following query is a depth-first enumeration by the left interval boundary:

```
SELECT depth, name, (numer_lft/denom_lft) AS indentation
FROM Hierarchy
ORDER BY indentation;
```

Results

depth	name
0	'King'
1	'Clark'
2	'Miller'
1	'Blake'
2	'Turner'
2	'Martin'
2	'Ward'
2	'Allen'
1	'Jones'
2	'Ford'
3	'Smith'
2	'Scott'
3	'Adams'

5.4.7 Depth-first Enumeration by Right Interval Boundary

Depth-first enumeration, ordering by right interval boundary, is demonstrated in the following query:

```
SELECT depth, name,
     (numer_rgt/denom_rgt) AS indentation
  FROM Hierarchy
ORDER BY indentation DESC;
```

Results

depth	name
0	'King'
1	'Jones'
2	'Scott'
3	'Adams'
2	'Ford'
3	'Smith'
1	'Blake'
2	'Allen'
2	'Ward'
2	'Martin'
2	'Turner'
1	'Clark'
2	'Miller'

You can get the same results by ordering by path.

```
SELECT depth, name, path
  FROM Hierarchy
 ORDER BY path;
```

5.4.8 All Descendants of a Node

Using 'Ford' as the ancestor in question and excluding him, the query is:

```
SELECT H2.name
 FROM Hierarchy AS H1, Hierarchy AS H2
WHERE H1.name = 'Ford'
  AND Distance (H1.numer, H1.denom, H2.numer, H2.denom) > 0;
```

Results

name
'King'
'Jones'

You can change the "> 0" to ">= 0" in the predicate if you wish to get the entire subtree rooted at the 'Ford' node.

This page intentionally left blank

The Linear Version of the Nested Sets Model

IF YOU LOOK at the diagram that shows the left and right numbers on a number line, you will realize that this diagram can be used directly to represent a tree in a nested sets model. The left and right (lft, rgt) numbers each appear once, but the nodes of the tree appear exactly twice—once with the lft number and once with the rgt number. The table can be defined like this:

```
CREATE TABLE OrgChart
(emp CHAR(10) NOT NULL,
seq INTEGER NOT NULL UNIQUE,
CONSTRAINT natural_numbers
     CHECK(seq > 0),
CONSTRAINT got_all_numbers
CHECK ((SELECT COUNT(*) FROM OrgChart)
        = (SELECT MAX(seq) FROM OrgChart)),
CONSTRAINT exactly_twice
CHECK (NOT EXISTS
        (SELECT *
           FROM OrgChart
          GROUP BY emp
         HAVING COUNT(*) <> 2)),
PRIMARY KEY (emp, seq));
```

In fairness the "got_all_numbers" and "exactly_twice" constraints will be hard to implement in most SQL products today, but they are legal in full SQL-92. Our OrgChart tree is represented by this data.

OrgChart

emp	seq
'Albert'	1
'Bert'	2
'Bert'	3
'Chuck'	4
'Donna'	5
'Donna'	6
'Eddie'	7
'Eddie'	8
'Fred'	9
'Fred'	10
'Chuck'	11
'Albert'	12

The standard nested sets model can be constructed using this VIEW for queries, which cannot be updated:

```
CREATE VIEW OrgChart_NS (emp, lft, rgt)
AS SELECT emp, MIN(seq), MAX(seq)
     FROM OrgChart
    GROUP BY emp;
```

Why bother with this approach? It can be handy for parsing mark-up language data into a relational table. You add a row for every begin tag and every end tag that you find as you read the text from left to right. There is a group looking into standards for SQL/XML, but at this writing we are very close to a generally accepted standard, both de facto and de jure.

6.1 Insertion and Deletion

Insertion and deletion are just modifications of the routines used in the standard nested sets model. For example, to remove a subtree rooted at :my_employee, you would use:

```
CREATE PROCEDURE RemoveSubtree (IN my_employee CHAR(10))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
 DECLARE leftmost INTEGER;
 DECLARE rightmost INTEGER;
 -- remember where the subtree root was
 SET leftmost = (SELECT MIN(seq)
                  FROM OrgChart
                 WHERE emp = my_employee);
SET rightmost = (SELECT MAX(seq)
                  FROM OrgChart
                 WHERE emp = my_employee);
-- remove the subtree
DELETE FROM OrgChart
 WHERE seq BETWEEN leftmost AND rightmost;
-- compute the size of the subtree & close the gap
UPDATE OrgChart
   SET seq = seq - (rightmost - leftmost + 1) / 2
   WHERE seq > leftmost;
END;
```

Insertion is the reverse of this operation. You must create a gap and then add the new subtree to the table.

```
CREATE PROCEDURE InsertSubtree (IN my_boss CHAR(10))
LANGUAGE SQL
DETERMINISTIC
BEGIN ATOMIC
-- assume that the new subtree is held in NewTree
-- and is in linear nested set format
DECLARE tree_size INTEGER;
DECLARE boss_right INTEGER;
-- get size of the subtree
SET tree_size = (SELECT COUNT(*) FROM NewTree);
-- place new tree to right of siblings
SET boss_right = (SELECT MAX(seq)
                    FROM OrgChart
                   WHERE emp = my_boss);
 -- move everyone over to the right
```

```
UPDATE OrgChart
   SET seq = seq + tree_size
  WHERE seq >= boss_right;
 -- re-number the subtree and insert it
 INSERT INTO OrgChart
 SELECT emp, (seq + boss_right) FROM NewTree;
 -- clear out subtree table
 DELETE FROM Subtrees;
END;
```

6.2 Finding Paths

The path from a node to the root can be found by first looking for the seq number that would represent the lft number of the node in the nested sets model, then returning the seq numbers lower than that value.

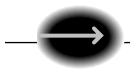
```
SELECT P1.emp
  FROM OrgChart AS P1
WHERE P1.seq <= (SELECT MIN(P2.seq) -- left parentheses
                   FROM OrgChart AS P2
                  WHERE P2.emp = :my_guy)
GROUP BY emp
HAVING COUNT(*) = 1;
```

This is a "flatten" version of the BETWEEN predicate in the nested sets model. The HAVING clause will remove pairs of siblings, leaving only the path.

6.3 Finding Levels

Getting the level is a little trickier. You count the "parentheses" (i.e., seq), then count the number of distinct things inside the parentheses (emp); every pair of parentheses moves you up a level. Then you do some algebra and come up with this answer.

```
SELECT :my_guy,
       2 * COUNT(DISTINCT P2.emp)
         - COUNT(DISTINCT P2.seq) AS lvl
 FROM OrgChart AS P1, OrgChart AS P2
WHERE P1.emp = :my_guy
 AND P2.seq <= (SELECT MIN(seq)
                  FROM OrgChart
                 WHERE emp = :my_guy);
```



6.4 Summary

Frankly, there is not much purpose in recommending this model. The only situation in which it might be useful would entail a data feed in which the closing tag arrives later than the starting tag.

This page intentionally left blank

Binary Trees

BINARY TREES ARE a special case of trees in which each parent can have, at most, only two children that are ordered (e.g., no children, a left child, a right child, or both a left and a right child). Binary trees are the subject of a lot of chapters in data structures books because they have such nice mathematical properties. For example, the number of distinct binary trees with (n) nodes is called a Catalan number, and it is given by the formula ((2n)!/((n + 1)!n!)). Let's stop and define some terms before we go any further.

Complete binary tree: a binary tree in which all leaf nodes are at level (n) or (n - 1), and all leaves at level (n) are toward the left, with "holes" on the right. There are between (2^(n - 1)) and ((2^n) - 1) nodes, inclusively, in a complete binary tree. A complete binary tree is efficiently implemented as an array, where a node at location (i) has children at indices (2 * i) and ((2 * i) + 1) and a parent at location (i/2). This is also known as heap and is used in the HeapSort algorithm, which we will get to later in this chapter.

Perfect binary tree: a binary tree in which each node has exactly zero or two children and all leaf nodes are at the same level. A perfect binary tree has exactly ((2^h) - 1) nodes, where (h) is the height. Every perfect binary tree is a full binary tree and a complete binary tree.

Balanced binary tree: a binary tree in which no leaf is more than a certain amount farther from the root than any other leaf. (See also AVL tree,

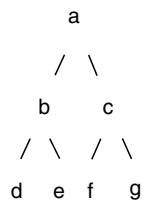
height-balanced tree, weight-balanced tree, B-tree, Red-Black tree and more in the literature.)

AVL tree: a balanced binary tree in which the heights of the two subtrees rooted at a node differ from each other by, at most, one. The structure is named for the inventors, Adelson-Velskii and Landis (1962).

Height-balanced tree: a tree whose subtrees differ in height by no more than one, and the subtrees are height-balanced as well. An empty tree is height-balanced. A binary tree can be skewed to one side or the other. As an extreme example, imagine a binary tree with only left children, all in a straight line. The ideal situation is to have a balanced binary tree—one that is as shallow as possible, because at each subtree the left and right children are the same size or no more than one node different. This will give us a worst search time of LOG2(n) tries for a set of (n) nodes.

Fibonacci tree: a variant of a binary tree in a tree of order (n) in which (n>1) has a left subtree of order (n -1) and a right subtree of order (n - 2). An order 0 Fibonacci tree has no nodes, and an order 1 tree has one node. A Fibonacci tree of order (n) has (F(n + 2) − 1) nodes, in which F(n) is the nth Fibonacci number. A Fibonacci tree is the most unbalanced AVL tree possible.

In the following example 'b' is the left son of 'a,' and 'c' is the right son of 'a.' Because all the locations have a value, this is called a complete binary tree.



In procedural programming languages, binary trees are usually represented with pointer chains or in one-dimensional array, where the array subscript determines the relationship the node holds within the tree structure. The array location is determined by the rule that if a node has array location of (n), then its left child has an array location of (2 * n) and its right child has an array location of ((2 * n) + 1). With a little algebra you can see that the parent of a node is FLOOR(n/2).

A binary tree is used for searching by placing data in the nodes in such a way that for every node in the tree, all the nodes in its left subtree are less than the parent node's value and all the nodes in its right are greater than the parent node's value. You locate a value by starting at the root of the tree and turning left or right, as required, until you find the value or that the value is not in the tree. All tree-indexing schemes, such as B - trees and B + trees, generalize this idea to a traversal in a multiway tree.

7.1 Binary Tree Traversals

One of the standard programs you have to write in freshman computer science is a traversal for a binary tree. A traversal is an orderly way of visiting every node so that you can perform some operation on it. There are three ways to traverse a binary tree, starting at the root.

1. Postorder traversal

a. Recursively traverse the left son's subtree,
b. Recursively traverse the right son's subtree, and
c. Visit the root of the current subtree

In the following sample tree you would get the list ('B', 'E', 'D', 'F', 'G', 'C', 'A'). This algorithm can be generalized to nonbinary trees, and it is called a depth-first search. If you were given the parse tree for an infixed arithmetic expression, as shown here, the postorder traversal would give you the reverse Polish notation equivalent of the expression.

```
      +

    /   \    = (2 3 +)

   2     3
```

This algorithm can be generalized to nonbinary trees, and it is called a breadth-first search.

2. Preorder traversal

a. Visit the root of the current subtree,
b. Recursively traverse the left son's subtree, and
c. Recursively traverse the right son's subtree

In the following sample tree you would get the list ('A', 'B', 'D', 'E', 'C', 'F', 'G'). This algorithm can be generalized to nonbinary trees, and it is

called a depth-first search. If you were given the parse tree for an infixed arithmetic expression, as shown here, the preorder traversal would give you the Polish notation equivalent of the expression.

```
        +

      /   \    = (+ 2 3)

     2     3
```

This algorithm can be generalized to nonbinary trees, and it is called a breadth-first search.

3. Inorder traversal

a. Recursively traverse the left son's subtree,
b. Visit the root of the current subtree, and
c. Recursively traverse the right son's subtree

In this sample tree you would get the list ('D', 'B', 'E', 'A', 'F', 'C', 'G'). If you were given the parse tree for an arithmetic expression, as shown here, the inorder traversal would give you the standard infixed notation equivalent of the expression.

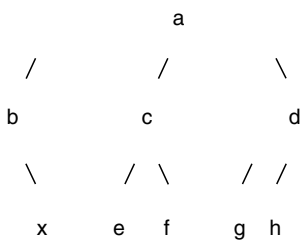
```
        +

      /   \    = (2 + 3)

     2     3
```

This algorithm does not generalize to nonbinary trees. Damjan S. Vujnovic (email: damjan@galeb.etf.bg.ac.yu) points out that the preorder and postorder representations work because there exists, at most, one tree that matches a given set of values. The inorder traversal situation is somewhat different. Consider the following two trees (nodes 'b' and 'c' are left children of node 'a'; node 'd' is right child of node 'a', and so on. Nodes having a "/" above are left children, and nodes having a "\" are right children):

MultiTree A:

```
                    a

          /         /         \

         b          c          d

               /   /  \     /   /

              x   e    f   g    h
```

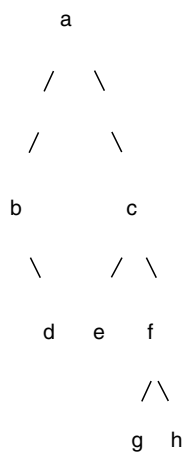
MultiTree B:



If we try to represent these trees using an inorder traversal, we find that they share the same representation; notice node 'X' in the diagrams. Therefore the inorder traversal works only with binary trees.

7.2 Binary Tree Queries

Vujnovic worked out the details of the following queries against a binary tree. Let's construct a binary tree and load it with some sample data.




```
CREATE TABLE BinTree
(node CHAR(10) NOT NULL,
 location INTEGER NOT NULL PRIMARY KEY);

INSERT INTO BinTree(node, location)
VALUES ('a', 1), ('b', 2), ('c', 3), ('d', 5),
    ('e', 6), ('f', 7), ('g', 14), ('h', 15);
```

The following table is useful for doing queries on the heap table:

```
CREATE TABLE PowersOfTwo
(exponent INTEGER NOT NULL PRIMARY KEY
          CHECK(exponent >= 0),
pwr_two INTEGER NOT NULL UNIQUE
          CHECK(pwr_two >= 1)
--, CHECK(2^exponent = pwr_two), but this is not standard SQL
);

INSERT INTO PowersOfTwo VALUES (0, 1);
INSERT INTO PowersOfTwo VALUES (1, 2);
INSERT INTO PowersOfTwo VALUES (2, 4);
INSERT INTO PowersOfTwo VALUES (3, 8);
INSERT INTO PowersOfTwo VALUES (4, 16);
INSERT INTO PowersOfTwo VALUES (5, 32);
INSERT INTO PowersOfTwo VALUES (6, 64);
INSERT INTO PowersOfTwo VALUES (7, 128);
INSERT INTO PowersOfTwo VALUES (8, 256);
```

Most SQL has base ten or natural logarithm functions, and LOG2() can be expressed using either of them. The general formulas, carried to more precision than most computers can handle, are:

```
LOG10(x)/LOG10(2) = LOG10(x)/0.30102999566398119521373889472449

LN(x)/LN(2) = LN(x)/0.69314718055994530941723212145818
```

7.2.1 Find Parent of a Node

Getting the parent of a given child is trivial:

```
SELECT BinTree.*, :my_child
FROM BinTree
WHERE location
=(SELECT FLOOR(location/2) AS parent
       FROM BinTree T1
     WHERE T1.node = :my_child)
```

Likewise, we know that the root of the whole tree is always at location one.

7.2.2 Find Subtree at a Node

Finding a subtree rooted at a particular node is a little more complicated. Note that the locations of the children of a node with location (n) are:

```
(2 * n), (2 * n) + 1
(4 * n), ..., (4 * n) + 3
(8 * n), ..., (8 * n) + 7
(16 * n), ..., (16 * n) + 15
...
```

The node with location (s) is a subordinate of a node with location (n) if (and only if) there exists (k), such that:

```
(2^k) * n <= s < (2^k) * (n + 1)
```

We know that (k) exists, therefore k = FLOOR (LOG2(s/n))
In other words, if:

```
s < (2^FLOOR(LOG2(s/n))) * (n + 1)
```

then the node with location (s) is a subordinate of a node with location (n). This is easier to see with an example:

```
Example one:
n = 3, s = 13
13 < (2^2)* 4
13 < 16
TRUE

Example two:
n = 2, s = 12
12 < (2^2) * 3
12 < 12
FALSE
```

Thus we have the subordinates query:

```
SELECT :my_root, T1.*
 FROM BinTree AS T1, BinTree AS T2
WHERE T2.node = :my_root
  AND T1.location
  < (FLOOR(LOG2(T1.location/T2.location))^2) * (n + 1);
```

This predicate lets you test a location number (j) to see if it is a descendant of the node with location number (k) at level (i).

```
j BETWEEN((2^i) * k) AND((2^i) * k + i)
```

To get all of the descendants you could use a table of sequential integers that includes integer from one to at least the depth of the tree. This method can be generalized for n-ary tree with a bit of algebra. If the value of (n) is known in advance, we could improve its performance by adding the node level as another column.

7.3 Deletion from a Binary Tree

Deletion of a leaf node from the binary tree is easy. Remove the row with the target node and leave the rest of the tree alone. Deleting a subtree requires using the subordinates query, thus:

```
DELETE FROM BinTree
WHERE node = :my_root
   AND location
       IN (SELECT T1.location
             FROM BinTree AS T1
            WHERE T1.location
                  < (FLOOR (LOG2(T1.location/BinTree.location))^2)
* (n + 1));
```

Deleting a node with subordinates requires a business rule about promotion of the subordinates because every node must have a parent. This depends on the individual case, and I cannot give a general statement about it.

7.4 Insertion into a Binary Tree

Insertion into the binary tree is easy if there is a vacant position in the tree. In general, new nodes are added as the left child then the right child of the target parent node. If all child positions are full, the tree must be reorganized according to some business rule.

7.5 Heaps

One of the nice things about a binary tree is that its predictable growth pattern allows you to assign a single number to locate each node. Sequentially number the nodes across the levels in the tree from left to right. This structure is also

known as a heap when it is presented in an array and is the basis for the HeapSort algorithm.

Therefore given a root node located at location (1), you know that its sons are at locations (2) and (3). Likewise, using integer division, the parent of a node is at location (n/2), and therefore the grandparent is at ((n/2)/2) = (n/4). This leads to a recurrence relation based on powers of two.

```
CREATE TABLE Heap
(node CHAR(10) NOT NULL,
 location INTEGER NOT NULL PRIMARY KEY);

INSERT INTO Heap VALUES ('A', 1);
INSERT INTO Heap VALUES ('B', 2);
INSERT INTO Heap VALUES ('C', 3);
INSERT INTO Heap VALUES ('D', 4);
INSERT INTO Heap VALUES ('E', 5);
INSERT INTO Heap VALUES ('F', 6);
INSERT INTO Heap VALUES ('G', 7);
INSERT INTO Heap VALUES ('H', 8);
```

The following table is useful for doing queries on the heap table:

```
CREATE TABLE PowersOfTwo
(exponent INTEGER NOT NULL PRIMARY KEY
          CHECK(exponent >= 0),
pwr_two INTEGER NOT NULL UNIQUE
          CHECK (pwr_two >= 1)
--, CHECK (2^exponent = pwr_two), but this is not standard SQL
);

INSERT INTO PowersOfTwo VALUES (0, 1);
INSERT INTO PowersOfTwo VALUES (1, 2);
INSERT INTO PowersOfTwo VALUES (2, 4);
INSERT INTO PowersOfTwo VALUES (3, 8);
INSERT INTO PowersOfTwo VALUES (4, 16);
INSERT INTO PowersOfTwo VALUES (5, 32);
INSERT INTO PowersOfTwo VALUES (6, 64);
INSERT INTO PowersOfTwo VALUES (7, 128);
INSERT INTO PowersOfTwo VALUES (8, 256);
```

In actual SQL products you might want to use base two logarithms (LOG2(n) = LOG(n)/LOG(2.0) = LOG(n)/0.69314718055994529) or user-

defined functions to check that the PowersOfTwo rows are correct. The LOG() and FLOOR() functions are not actually part of Standard SQL, but they are common enough to be portable.

Given a table with powers of two, we can find all the ancestors of a node with this query, which depends on integer division.

```
SELECT H1.node, H1.location
 FROM Heap AS H1
WHERE H1.location
      IN (SELECT @my_node/pwr_two
            FROM PowersOfTwo
           WHERE pwr_two <= :my_location);
```

The level of a node is easy because each level starts with a power of two on the left side. (Remember that "level" is a reserved word in SQL-99.)

```
SELECT location, CAST (FLOOR(LOG(location)/LOG(2.0)) AS INTEGER)
AS lvl
 FROM Heap;
```

The depth of the heap is much the same, but it must include the incomplete level. Therefore it is the maximum level or the maximum level plus one.

```
(SELECT CAST (FLOOR(LOG(MAX(location))/LOG(2.0)) + 1.0 AS
INTEGER)
   FROM Heap) AS depth;
```

Finding the descendents is much harder. Here is a solution from John Gilson (email: jag@acm.org), who also provided the two previous queries:

```
CREATE VIEW HeapDescendants
(node, location, descendant, dscnt_loc)
AS
SELECT H1.node, H1.location,
       H2.node AS dscnt,
       H2.location AS dscnt_loc
  FROM (SELECT FLOOR(LOG(MAX(location))/LOG(2.0)) + 1.0
          FROM Heap) AS D(depth)
        CROSS JOIN
       (SELECT location, FLOOR(LOG(location)/LOG(2.0))
```

```
   FROM Heap) AS L(location, lvl)
INNER JOIN
Heap AS H1
ON H1.location = L.location
INNER JOIN
PowersOfTwo AS T
ON T.exponent >= 0
    AND T.exponent < D.depth - L.lvl
INNER JOIN
Heap AS H2
ON H2.location >= H1.location * pwr_two
    AND H2.location < H1.location * pwr_two + pwr_two;
```

Given the sample table, we would get this result:

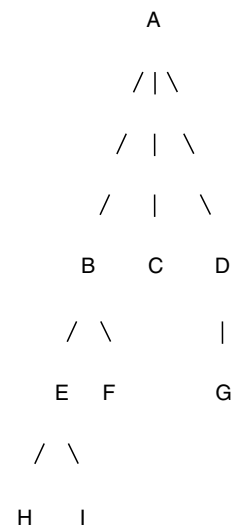
Results

node	location	dscnt	dscnt_loc
'A'	1	'A'	1
'A'	1	'B'	2
'A'	1	'C'	3
'A'	1	'D'	4
'A'	1	'E'	5
'A'	1	'F'	6
'A'	1	'G'	7
'A'	1	'H'	8
'B'	2	'B'	2
'B'	2	'D'	4
'B'	2	'E'	5
'B'	2	'H'	8
'C'	3	'C'	3
'C'	3	'F'	6
'C'	3	'G'	7
'D'	4	'D'	4
'D'	4	'H'	8
'E'	5	'E'	5
'F'	6	'F'	6
'G'	7	'G'	7
'H'	8	'H'	8

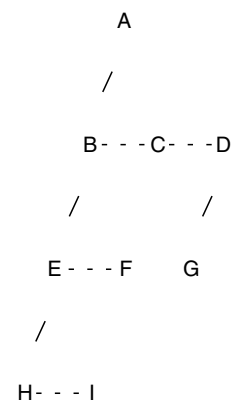
7.6 Binary Tree Representation of Multiway Trees

There is a simple way to represent a multiway tree as a binary tree. The algorithm is given in *Knuth's Art of Programming* (Donald E. Knuth, Addison-Wesley, vol 1, Section 2.3.2, 1997). The binary tree representation of a multiway tree is based on first child-next sibling representation of the tree. In this representation every node is linked with its leftmost child and its next (right nearest) sibling.

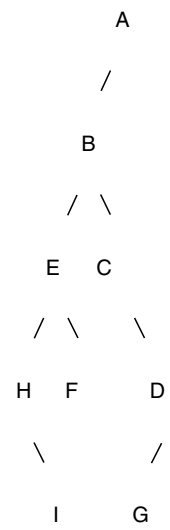
Informally, you take the original tree and traverse the nodes by going down a level, then across the siblings. The leftmost sibling (if any) becomes the left child in the binary tree. The sibling in the second position (if any) becomes the right child in the binary, and the third and younger siblings become right children under the second child. The algorithm is applied recursively down the tree. If you see one example, you will understand the idea. Let's start with this multiway tree:



This tree can be represented in first child-next sibling manner, as follows:



Now grab this graph and pull it up a little so that things flop down 45 degrees. Yes, that is not a very scientific description, but it makes good visual sense, doesn't it?



Behold! A binary tree!

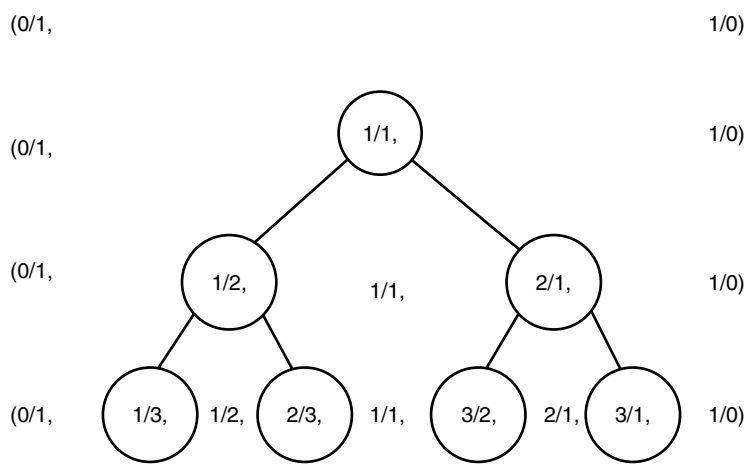
This example is credited to Paul E. Black (email: paul.black@nist.gov) and it's part of the dictionary of algorithms from NIST (website: www.nist.gov/dads/HTML/binaryBinTreeRepofBinTree.html).

The left child of a node is its immediate oldest subordinate, and the chain of right children from this root node is the other subordinates in order by age (i.e., left to right).

7.7 The Stern-Brocot Numbers

This is a method for constructing the set of all nonnegative fractions, (m/n), in which m and n are relatively prime. It also represents any binary tree by assigning a unique fraction to each node.

The process begins with the pair of fractions (0/1, 1/0), then you insert the fraction (m1 + m2)/(n1 + n2) between each pair of fractions (m1/n1, m2/n2). For example, the first steps in the process give us:



Remove (0/1) (i.e., zero) and (1/0) (i.e., infinity) and leave (1/1) (i.e., one) as the root of a binary tree. This maps every rational number into a set of left-right paths. For example, we can arrive at (5/7) by traversing the tree (left, right, right, left). It is a bit of algebra and programming, but you can map any tree into a binary tree, and then use the Stern-Brocot numbering to identify the nodes. Unfortunately, finding relationships in such a representation also requires a bit of algebra and programming.

Other Models for Trees

THE MODELS FOR trees and hierarchies discussed so far are not the only ones. There are other models that use different approaches and properties of trees, some of which are hybrids of other models.

8.1 Adjacency List with Self-references

A slight modification of the usual adjacency list model is to include an edge that loops back to the same node.

```
CREATE TABLE Personnel_OrgChart
(boss VARCHAR(20) NOT NULL,
 emp VARCHAR(20) NOT NULL,
 PRIMARY KEY (boss, emp));
```

Personnel_OrgChart

boss	emp
'Albert'	'Albert'
'Albert'	'Bert'
'Albert'	'Chuck'
'Bert'	'Bert'
'Chuck'	'Chuck'

(cont.)

boss	emp
'Chuck'	'Donna'
'Chuck'	'Eddie'
'Chuck'	'Fred'
'Donna'	'Donna'
'Eddie'	'Eddie'
'Fred'	'Fred'

This makes the table longer, but it avoids a NULL in the boss column of the root. The query for finding the leaf nodes is:

```
SELECT P1.boss
  FROM Personnel_OrgChart AS P1
 GROUP BY P1.boss
HAVING COUNT (P1.boss) = 1;
```

The other queries for the adjacency list still work in a modified form, but produce slightly different results.

8.2 Subordinate Adjacency List

Another modification of the usual adjacency list model is to show the edges of the graph as oriented from the superior to the subordinate. Nodes without a subordinate are leaf nodes, and they have a NULL.

```
CREATE TABLE Personnel_OrgChart
(emp VARCHAR(20) NOT NULL,
 subordinate VARCHAR(20), -- null means leaf node
 PRIMARY KEY (boss, emp));
```

Personnel_OrgChart

emp	subordinate
'Albert'	'Bert'
'Bert'	NULL
'Albert'	'Chuck'
'Chuck'	'Donna'
'Chuck'	'Eddie'
'Chuck'	'Fred'
'Donna'	NULL

(cont.)

emp	subordinate
'Eddie'	NULL
'Fred'	NULL

This avoids a NULL in the root, but gives you more NULLs in the table. Finding all the leaf nodes is easy:

```
SELECT P1.emp
 FROM Personnel_OrgChart AS P1
WHERE P1.subordinate IS NULL;
```

The queries for the adjacency list model still work, but they need modifications.

8.3 Hybrid Models

It is possible to mix the models we have discussed. The idea is to gain the advantages of each in one table, but the price can be increased overhead and storage.

8.3.1 Adjacency and Nested Sets Model

This approach retains the parent node column in each row of a nested sets model. The problem is that you cannot include the constraints on the left and right (lft, rgt) pairs that assure the tree structure, thus:

```
CREATE TABLE Tree
(node CHAR(5) NOT NULL,
 parent_node CHAR(5),
 lft INTEGER DEFAULT 0 NOT NULL,
 rgt INTEGER DEFAULT 0 NOT NULL);

INSERT INTO Tree
VALUES ('A', NULL, 1, 18),
       ('B', 'A', 2, 3),
       ('C', 'A', 4, 11),
       ('D', 'C', 5, 6),
       ('E', 'C', 7, 8),
       ('F', 'C', 9, 10),
       ('G', 'A', 12, 17),
       ('H', 'G', 13, 14),
       ('I', 'G', 15, 16);
```

The advantage of this model is that you can insert nodes using this statement and let the default values take effect.

```
INSERT INTO Tree (node, parent)
VALUES (:my_node, :my_parent);
```

The clean-up procedure has to detect any (0, 0) pairs in the tree table. If there is at least one such pair, we know nodes have been added. Therefore the procedure needs to perform a complete rebuild of the tree from the (child, parent) columns. If there is no such pair, we know that nodes might have been deleted, so the procedure needs to rerenumber the (lft, rgt) columns.

8.3.2 Nested Set with Depth Model

This approach retains the level or depth in each row of a nested sets model, disregarding constraints, thus:

```
CREATE TABLE Tree
(node CHAR(5) NOT NULL,
 "depth" INTEGER NOT NULL, -- depth is reserved in SQL - 99
 lft INTEGER NOT NULL,
 rgt INTEGER NOT NULL);

INSERT INTO Tree
VALUES ('A', 1, 1, 18),
       ('B', 2, 2, 3),
       ('C', 2, 4, 11),
       ('D', 3, 5, 6),
       ('E', 3, 7, 8),
       ('F', 3, 9, 10),
       ('G', 2, 12, 17),
       ('H', 3, 13, 14),
       ('I', 3, 15, 16);
```

Although the level number can be generated from the nested sets model in a VIEW, the query involves an expensive self-join. The advantage is in bill of materials (B.O.M. or BOM) problems in which subassembly data has to be computed up the tree from the leaf nodes (parts).

8.3.3 Adjacency and Depth Model

This model adds a column for the depth of the node to the adjacency list, thus:

```
CREATE TABLE Tree
(node CHAR(5) NOT NULL PRIMARY KEY,
 parent CHAR(5),
 "depth" INTEGER NOT NULL, -- depth is reserved in SQL-99
 CHECK (...), -- constraints for tree structure
);
```

Adding a node is easy:

```
CREATE PROCEDURE AddChildNode (IN c INTEGER, IN p INTEGER)
DETERMINISTIC
LANGUAGE SQL
INSERT INTO Tree
SELECT c, p, ("depth" + 1)
 FROM Tree
WHERE node = p;
```

However, this is a bad hybrid if you need to change the tree structure. When you delete a node, all the elements of its subtree have to be raised one level. Likewise, the depth has to be recalculated if a node is moved to a new parent. Tracing the path down the tree can be expensive in the adjacency list model because you need procedural code.

8.3.4 Computed Hybrid Models

John Gilson (jag@acm.org) came up with this set of VIEWs. For a given node N and a depth-first (preorder) traversal, each ancestor's sequence number is the greatest number on that level that is less than N's sequence number. For a given node N and breadth-first (postorder) traversal, each ancestor's sequence number is the least number on that level that is greater than N's sequence number. We can use these relationships directly to define the following VIEWS:

```
CREATE TABLE PreorderTree
(node VARCHAR(10) NOT NULL PRIMARY KEY,
 postorder_nbr INTEGER NOT NULL CHECK (postorder_nbr > 0),
 lvl INTEGER NOT NULL CHECK (lvl > 0),
UNIQUE (lvl, postorder_nbr));


-- Preorder
INSERT INTO PreorderTree
```

```
VALUES ('A', 1, 1),
       ('B', 2, 2),
       ('C', 3, 2),
       ('D', 4, 3),
       ('E', 5, 3),
       ('F', 6, 3),
       ('G', 7, 2),
       ('H', 8, 3),
       ('I', 9, 3);

CREATE VIEW PreorderRelationships
AS
SELECT T1.node AS descendant,
       T1.lvl AS descendant_lvl,
       T1.postorder_nbr AS descendant_postorder_nbr,
       T2.node AS ancestor,
       T2.lvl AS ancestor_lvl,
       T2.postorder_nbr AS ancestor_postorder_nbr
FROM PreorderTree AS T1
     INNER JOIN
     PreorderTree AS T2
     ON T2.lvl < T1.lvl
        AND T2.postorder_nbr < T1.postorder_nbr
     LEFT OUTER JOIN
     PreorderTree AS T3
     ON T3.lvl = T2.lvl
        AND T3.postorder_nbr > T2.postorder_nbr
        AND T3.postorder_nbr < T1.postorder_nbr
WHERE T3.postorder_nbr IS NULL;
```

Likewise for a postorder traversal:

```
CREATE TABLE PostorderTree
(node VARCHAR(10) NOT NULL PRIMARY KEY,
 postorder_nbr INTEGER NOT NULL CHECK (postorder_nbr > 0),
 lvl INTEGER NOT NULL CHECK (lvl > 0),
 UNIQUE (lvl, postorder_nbr));

-- Postorder
INSERT INTO PostorderTree
```

```
VALUES ('A', 9, 1),
       ('B', 1, 2),
       ('C', 5, 2),
       ('D', 2, 3),
       ('E', 3, 3),
       ('F', 4, 3),
       ('G', 8, 2),
       ('H', 6, 3),
       ('I', 7, 3);

CREATE VIEW PostorderRelationships
AS
SELECT T1.node AS descendant,
       T1.lvl AS descendant_lvl,
       T1.postorder_nbr AS descendant_postorder_nbr,
       T2.node AS ancestor,
       T2.lvl AS ancestor_lvl,
       T2.postorder_nbr AS ancestor_postorder_nbr
FROM PostorderTree AS T1
     INNER JOIN
     PostorderTree AS T2
     ON T2.lvl < T1.lvl
        AND T2.postorder_nbr > T1.postorder_nbr
     LEFT OUTER JOIN
     PostorderTree AS T3
     ON T3.lvl = T2.lvl
        AND T3.postorder_nbr < T2.postorder_nbr
        AND T3.postorder_nbr > T1.postorder_nbr
WHERE T3.postorder_nbr IS NULL;
```

We can then easily write some of the standard queries. Using the preorder tree, get all ancestors of a given node.

```
SELECT *
 FROM PreorderRelationships
WHERE descendant = :my_guy;
```

Using postorder, get all descendants of C:

```
SELECT *
FROM PostorderRelationships
WHERE ancestor = :my_ancestor;
```

8.4 General Graphs

For years I had been trying to find a clever trick to use some version of the nested sets model to represent more general graphs in SQL and I had no real luck. The problem is fundamental. Trees are planar graphs; that is, they can be drawn on a Cartesian plane without having any of the lines cross over one another. The nested sets model essentially points on a Cartesian plane (x, y) with a partial order defined by:

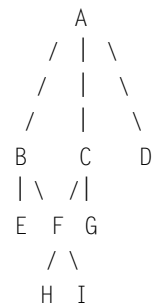
```
((x1, y1) <= (x2, y2)) <=> ((x1 <= x2) AND (y1 <= y2))
```

Now we can see why nested sets cannot model arbitrary directed acyclic graphs: two dimensions are not sufficient for representing arbitrary partial orders. The only way to represent a directed acyclic graph in SQL seems to be to use an adjacency list model without the constraints that would force the graph to be a tree.

8.4.1 Detecting Paths in a Convergent Graph

Let's build a simple graph with nine nodes, as shown on the next page, and represent it with its edges.

```
CREATE TABLE Graph
(in_node CHAR(1) NOT NULL,
out_node CHAR(1) NOT NULL,
PRIMARY KEY (in_node, out_node));
INSERT INTO Graph VALUES ('A', 'B');
INSERT INTO Graph VALUES ('A', 'C');
INSERT INTO Graph VALUES ('A', 'D');
INSERT INTO Graph VALUES ('B', 'E');
INSERT INTO Graph VALUES ('B', 'F');
INSERT INTO Graph VALUES ('C', 'F');
INSERT INTO Graph VALUES ('C', 'G');
INSERT INTO Graph VALUES ('F', 'H');
INSERT INTO Graph VALUES ('F', 'I');
```



This is a convergent graph, which means that it has a root (a node whose indegree is zero) and leaf nodes (a set of nodes whose outdegree is zero), but the other nodes have (indegree >=1) and (outdegree >=1). Informally, it is a tree with some crossover edges added, but the flow is still downhill and we have no directed cycles.

The easiest way to find all the paths in this graph is to build a path enumeration table. Start by building all paths of the length two by concatenating the edges given in the graph table. In a loop connect all valid pairs of paths into longer and longer paths. You stop when you cannot find any more new paths. We also know that the longest possible path is the number of nodes in the graph minus one.

```
CREATE TABLE Paths (path_string VARCHAR(500) NOT NULL);

CREATE PROCEDURE GraphPaths ()
DETERMINISTIC
LANGUAGE SQL
BEGIN
DECLARE counter INTEGER;
DECLARE old_counter INTEGER;
-- get a start in the Paths table:
DELETE FROM Paths;
INSERT INTO Paths(path_string)
SELECT in_node || out_node FROM Graph;

-- set up loop controls
SET counter = (SELECT COUNT(*) FROM Paths);
SET old_counter = 0;
```

```
-- loop while path set gets bigger
WHILE counter > old_counter
DO
INSERT INTO Paths(path_string)
SELECT DISTINCT P1.path_string || SUBSTRING (P2.path_string FROM
2 FOR 1)
  FROM Paths AS P1, Paths AS P2,
       Sequence AS S1
WHERE SUBSTRING (P1.path_string, CHAR_LENGTH(P1.path_string), 1)
= SUBSTRING (P2.path_string FROM 1 FOR 1)
     AND (P1.path_string || SUBSTRING (P2.path_string FROM 2 FOR
1))
           NOT IN (SELECT path_string FROM Paths)
     AND S1.postorder_nbr BETWEEN 3
                AND (SELECT COUNT(*) FROM Graph) - 1
     AND CHAR_LENGTH((P1.path_string || SUBSTRING (P2.path_string,
2, 1)))
           <= postorder_nbr
     AND NOT EXISTS
           (SELECT *
              FROM Graph AS G1
             WHERE CHAR_LENGTH(P1.path_string || SUBSTRING
(P2.path_string FROM 2 FOR 1))
                   - CHAR_LENGTH(REPLACE(P1.path_string ||
SUBSTRING (P2.path_string FROM 2 FOR 1), G1.in_node, ''))
> 1);

-- keep old tally and compute new tally
SET old_counter = counter;
SET counter = (SELECT COUNT(*)FROM Paths);
END WHILE; -- of loop
-- display Paths table: SELECT * FROM Paths ORDER BY
path_string;
END;
```

When you concatenate two paths, the head of one path has to match the tail of the other and you have to remember to cut off the head before doing the concatenation. Because we do not want to record cycles, we need to test to be sure that a path string does not have two copies of the same node

name in it. The REPLACE() function is not Standard SQL, but it is very common.

Paths

path_string
AB
ABE
ABF
ABFH
ABFI
AC
ACF
ACFH
ACFI
ACG
AD
BE
BF
BFH
BFI
CF
CFH
CFI
CG
FH
FI

If the nodes are numbered or longer than one character, cast them to strings of a known fixed length or use a separator. This makes the code a bit more complex, but does not really change the underlying ideas.

8.4.2 Detecting Directed Cycles

Let's use the same graph as in the previous section and add a new edge ('I', 'C'), which will create a cycle among nodes ('C', 'F', 'I'). How do we find cycles in such a graph?

The path detection algorithm given in the previous section will give all three traversals around two of the three edges of the ('C', 'F', 'I') cycle, as it should. This code will give you pairs of cycles.

```
SELECT P1.path_string, P2.path_string
 FROM Paths AS P1, Paths AS P2, Sequence AS S1
WHERE CHAR_LENGTH(P1.path_string) = CHAR_LENGTH(P2.path_string)
  AND SUBSTRING(P1.path_string FROM (postorder_nbr + 1)
                FOR CHAR_LENGTH(P1.path_string))
      || SUBSTRING(P1.path_string FROM 1 FOR postorder_nbr)
      = P2.path_string
  AND postorder_nbr <= CHAR_LENGTH(P1.path_string)
  AND P1.path_string < P2.path_string;
```

For two paths to be part of a cycle they first have to be of the same length. Using the sequence table, which is a table that contains the integers from 1 to (n), we can cut off all possible heads from one of the candidates and concatenate them to its corresponding tail. When one of these permutations matches the first path, they are in the same cycle.

Proprietary Extensions for Trees

As you can see from the examples given earlier in this book, you very quickly get into recursive or procedural code to handle trees. Because the single-table adjacency list model is popular, several vendors have added extensions and academics have proposals to handle tree traversal in SELECT statements.

9.1 Oracle Tree Extensions

Oracle has CONNECT BY PRIOR and START WITH clauses in the SELECT statement to provide partial support for reachability and path enumeration queries. The START WITH clause tells the engine the value of the root of the tree. The CONNECT BY PRIOR clause establishes the edges of the graph. The function LEVEL gives the distance from the root to the current node, starting at 1 for the root. Let's use a list of parts and subcomponents as the example database table. The query "Show all subcomponents of part A1, including the substructure" can be handled by the following Oracle SQL statement:

```
SELECT LEVEL AS path_length, assembly_nbr, subassembly_nbr
   FROM Blueprint
  START WITH assembly_nbr = 'A1'
CONNECT BY PRIOR subassembly_nbr = assembly_nbr;
```

The query produces the following result:

Result1

path_length	assembly_nbr	subassembly_nbr
1	'A1'	'A2'
2	'A2'	'A5'
2	'A2'	'A6'
3	'A6'	'A8'
3	'A6'	'A9'
1	'A1'	'A3'
2	'A3'	'A6'
3	'A6'	'A8'
3	'A6'	'A9'
2	'A3'	'A7'
1	'A1'	'A4'

The output is an adequate representation of the query result because it is possible to construct a path enumeration tree from it. The CONNECT BY PRIOR clause provides traversal, but not support, for recursive functions before version 9.0. (Check the status of the product at the time you are reading this chapter.) For example, it is not possible to sum the weights of all subcomponents of part A1 to find the weight of A1. The only recursive function supported by the CONNECT BY PRIOR clause is the LEVEL function. Another limitation of the CONNECT BY PRIOR clause is that it does not permit the use of joins. The reason for disallowing joins is that the order in which the rows are returned in the result is important. The parent nodes appear before their children, so you know that if the path length increases, these are children; if it does not, they are new nodes at a higher level.

This also means that an ORDER BY clause can destroy any meaning in the results. This means, moreover, that the CONNECT BY PRIOR result is not a true table; a table by definition does not have an internal ordering. In addition, this means that it is not always possible to use the result of a CONNECT BY query in another query. A trick for working around this limitation, which makes indirect use of the CONNECT BY PRIOR clause, is to hide it in a subquery that is used to make a JOIN at the higher level. For example, to attach a product category description, form another table to the parts explosion.

```
SELECT part_nbr, category_name
 FROM Parts, ProductCategories
WHERE Parts.category_id = ProductCategories.category_id
```

```
 AND part_nbr IN (SELECT subassembly_nbr
                    FROM Blueprint
                   START WITH assembly_nbr = 'A1'
                 CONNECT BY PRIOR subassembly_nbr = assembly_nbr);
```

Before Oracle 9i the subquery had to have only one table in the FROM clause to comply with the restriction that there must be no joins in any query block that contains a CONNECT BY PRIOR clause. On the other hand, the main query involves a JOIN of two tables, which would not be possible with direct use of the CONNECT BY PRIOR clause. Another query that cannot be processed by direct use of the CONNECT BY PRIOR clause is one that displays all parent-child relationships at all levels. A technique to process this query is illustrated by the following SQL:

```
SELECT DISTINCT PX.part_nbr, PX.pname, PY.part_nbr, PY.pname
 FROM Parts AS PX, Parts AS PY
WHERE PY.part_nbr
      IN (SELECT Blueprint.subassembly_nbr
            FROM Blueprint
          START WITH assembly_nbr = PX.part_nbr
          CONNECT BY PRIOR subassembly_nbr = assembly_nbr)
ORDER BY PX.part_nbr, PY.part_nbr;
```

Again, the outer query includes a JOIN, which is not allowed with the CONNECT BY PRIOR clause in the inner query used in the IN() predicate. Note that the correlated subquery references PX.part_nbr.

9.2 XDB Tree Extension

XDB Systems, which has been out of business for several years, was an SQL product that ran on PC platforms and was fully compatible with DB2. The company was founded by Dr. S. Bing Yao, who was known for his research in query optimization. The product has a set of extensions similar to those in Oracle, but this product uses functions rather than clauses to hide the recursion. The PREVIOUS (<column>) function finds the parent node value of the child column for the row being currently processed by a query. The keyword LEVEL is a system value computed for each row, which gives its path length from the root; the root is at LEVEL = 0. There is a special value for the path length of a leaf node, called BOTTOM. For example, to find all of the subcomponents of A1, you would write the following query:

```
SELECT assembly_nbr
 FROM Blueprint
WHERE PREVIOUS (subassembly_nbr) = assembly_nbr
  AND assembly_nbr = 'A1'
  AND LEVEL <= BOTTOM;
```

Other vendors have done similar things, all based on establishing a root and a relationship to JOIN the original table to a correlated copy of itself. Indexing can help, but such queries are still very expensive.

9.3 DB2 and the WITH Operator

IBM added the WITH operator from the SQL - 99 standard to their DB2 product line to handle the need to factor out common subquery expressions and give them a name for the duration of the query. The other alternatives have been to repeat the code (hoping that the optimizer would do the factoring) or to create a VIEW and use it. However, the VIEW will be persistent in the schema after the query is done, unless you explicitly drop it.

However, instead of being a simple temporary VIEW mechanism, IBM made the WITH clause handle recursive queries by allowing self-references. This is useful for tree structures in particular. You define a special form of the temporary hidden table that has an initial subquery and a recursive subquery. These two parts have to be connected by a UNION ALL operator—no other set operation will do. The hidden table is initialized with the results of the initial subquery, and then the result of the recursive subquery is added to the hidden table over and over as it is used. This might be easier to explain with an example, taken from the usual adjacency list model OrgChart table. To find the immediate subordinates of boss 'Albert' you would write:

```
SELECT *
 FROM OrgChart
WHERE boss = 'Albert';
```

To find all of his subordinates you add this WITH clause to the query:

```
WITH Subordinates (emp, salary)
AS (SELECT emp, salary
       FROM OrgChart AS PO
      WHERE boss = 'Albert') -- initial set
     UNION ALL
    (SELECT emp, salary
```

```
       FROM OrgChart AS P1, Subordinates AS S1
      WHERE P1.boss = S1.emp) -- recursive set
SELECT emp
  FROM Subordinates;
```

Each time you fetch a row from Subordinates, the WITH clause is executed using the current rows of the temporary hidden table. First, I fetch 'Albert' and his immediate subordinates. I then do a UNION ALL for the personnel who have those subordinates as bosses, and so forth until the subquery is empty. Then the hidden table is passed to the main SELECT clause to which the WITH clause is attached.

9.4 Date's EXPLODE Operator

In his book, *Relational Database: Selected Writings* (Addison-Wesley, ISBN 0-201-14196-5, 1986), Chris Date proposed an EXPLODE(<table name>) table valued function that would take an input table in the adjacency list model and return another table with four columns: the level number, the current node, the subordinate node, and the sequence number. The sequence number was included to get around the problem of the ordering's having meaning in the hierarchy. The EXPLODE results are derived from simple tree-traversal rules.

It is possible to write such a function in the current version of products that have a table-valued function feature. You can also write a procedure that will write the result set to a global or local temporary table that the rest of the session can use.

9.5 Tillquist and Kuo's Proposals

John Tillquist and Feng-Yang Kuo proposed an extension wherein a tree in an adjacency list model is viewed as a special kind of GROUP BY clause (Tillquist and Kuo, "An Approach to the Recursive Retrieval Problem in Relational Databases," Communications of the ACM. 32, 2 [February 1989]: 239). They would add a GROUP BY LEAVES (major, minor) that can be approximated with the following query:

```
SELECT *
 FROM Tree AS T1
WHERE NOT EXISTS (SELECT *
                    FROM Tree AS T2
                   WHERE T1.major = T2.minor)
GROUP BY T1.major;
```

The idea is that you get groups of leaf nodes, with their immediate parent as the single grouping column. Other extensions in Tillquist and Kuo's paper include a GROUP BY NODES (<parent node>, <child node>), which would use each node only once to prevent problems with cycles in the graph and would find all of the descendants of a given parent node. They then extend the aggregate functions with a COMPOUND function modifier (along the lines of DISTINCT) that carries the aggregation up the tree.

9.6 Microsoft Extensions

Microsoft defines a hierarchical rowset as one in which one of the columns of the parent rowset is itself another rowset, generated with this syntax.

```
SHAPE {SELECT au_id, au_lname, au_fname FROM authors}
APPEND ({SELECT au_id, title
           FROM titleauthor TA, titles TS
          WHERE TA.title_id = TS.title_id}
       AS title_chap RELATE au_id TO au_id)
```

You can find more details online at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dblibc/dtsptasks_8jqh.asp.

This facility and the shape/append command were not available as part of SQL Server, but only as part of Microsoft Data Shaping Service for OLE DB.

9.7 Other Methods

Looking at the literature, most of the attempts to add a tree structure operation to SQL have been based on the assumption that the adjacency list representation was the only possible way to model a tree structure. However, as we can see in this book, that simply is not true.

Perhaps the influence of decades of procedural languages is hard to overcome. Or it might be all of those "boxes-and-arrows" charts we have seen on the walls even before there were computers.

Hierarchies in Data Modeling

TYPE HIERARCHIES ARE useful when you are trying to model entities for a database. How this hierarchy is mapped into SQL DDL is another issue. Many years ago, at an ANSI X3H2 Database Standards Committee meeting in Rapid City, S.D., Bjarne Stroustrup gave a lecture on C++ and object-oriented (OO) programming research at Bell labs. When asked about using OO concepts in databases, he replied that the people at Bell labs had experimented with it, tried several approaches, and came to the conclusion that although OO was good for programming, it was a bad idea for data. The most recent model of OO also seems to have gone back to a separation of data and procedures.

However, programmers who come into SQL from OO languages and models insist on trying to model class or type hierarchies in SQL. This is not a new phenomenon. When SQL first came out, COBOL programmers tried to force their mental model on SQL. The old files were converted directly into tables, each field became a column, and each record became a row. Then the application program could simply replace the file reads with a cursor, and the programmer never had to learn the relational model. The performance stunk, of course.

The usual attempts by OO programmers to force their model into SQL involve building a metadata model in SQL, in which the tables use a proprietary, nonrelational, auto-incrementing feature of some kind to replace a global object identifier and have columns that contain the names of attributes, their values, and something to establish the class hierarchy.

These auto-numbering features are a holdover from the early SQL products, which were based on existing file systems. The data was kept in physically contiguous disk pages, in physically contiguous rows, made up of physically contiguous columns—just like a deck of punch cards or a magnetic tape.

However, physically contiguous storage is only one way of building a relational database, and it is not always the best option. One of the basic ideas of a relational database is that the user is not supposed to know anything about how things are stored, much less write code that depends on the particular physical representation in a particular release of a particular product. Because every underlying file system was different and there was no standard, every vendor came with a proprietary and nonportable scheme for auto-numbering.

Let's look at the logical problems of auto-numbering. First try to create a table with two columns, and try to make them both auto-numbered. Many products, such as the Sybase/SQL Server family and their IDENTITY, cannot declare more than one column of this pseudo-data type. Next create a table with one auto-numbered column. Now try to insert, update, and delete different numbers from it. If you cannot insert, update, and delete rows from a table, it is not a table by definition. These auto-numbers are a PHYSICAL property of the PHYSICAL table storage and have nothing to do with the data in the table or the LOGICAL model of the data.

It gets worse; create a simple table with one auto-numbered column and a few other columns. Insert a few rows into the table, thus letting the auto-numbered column that is not shown in the list default to its automatic values.

```
INSERT INTO Foobar (a, b, c) VALUES ('a1', 'b1', 'c1');
INSERT INTO Foobar (a, b, c) VALUES ('a2', 'b2', 'c2');
INSERT INTO Foobar (a, b, c) VALUES ('a3', 'b3', 'c3');
```

You will notice that the auto-numbering is sequential and in the order the INSERT INTO statements were presented. If you delete a row, the gap in the sequence is not filled in and the sequence continues from the highest number that has ever been used in that column in that particular table.

Now use an INSERT INTO statement with a query expression in it, like this:

```
INSERT INTO Foobar (a, b, c)
SELECT x, y, z
  FROM Floob;
```

Because a query result is a table, and a table is a set that has no ordering, what should the auto-numbers be? The whole, completed set is presented to

Foobar all at once, not a row at a time. There are (n!) ways to number (n) rows, so which one do you pick? The answer has been to use whatever the PHYSICAL order of the result set happened to be. It is actually worse than that. If the same query is executed again, but with new statistics or after an index has been dropped or added, the new execution plan could bring the result set back in a different PHYSICAL order. Can you explain from a LOGICAL modeling viewpoint why the same rows in the second query get different auto-numbers?

Using auto-numbering as a primary key is a sign that there is no data model, only an imitation of a sequential file system. Since this number exists only as a result of the state of a particular piece of hardware at a particular time in a particular release of a particular version of an SQL product, how do you verify such a number in the reality you are modeling?

To quote from Dr. Codd, "...Database users may cause the system to generate or delete a surrogate, but they have no control over its value, nor is its value ever displayed to them...." (Codd E: Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems* 4(4):397 - 434, 1979). This means that a surrogate ought to act like an index; created by the user, managed by the system, and NEVER seen by a user. Dr. Codd also wrote the following

"There are three difficulties in employing user-controlled keys as permanent surrogates for entities.

(1) The actual values of user-controlled keys are determined by users and must therefore be subject to change by them (e.g., if two companies merge, the two employee databases might be combined with the result that some or all of the serial numbers might be changed).

(2) Two relations may have user-controlled keys defined on distinct domains (e.g., one uses social security, while the other uses employee serial numbers) and yet the entities denoted are the same.

(3) It may be necessary to carry information about an entity either before it has been assigned a user-controlled key value or after it has ceased to have one (e.g., an applicant for a job and a retiree).

These difficulties have the important consequence that an equi-join on common key values may not yield the same result as a join on common entities. A solution—proposed in part [4] and more fully in [14]—is to introduce entity domains which contain system-assigned surrogates. Database users may cause the system to generate or delete a surrogate, but they have no control over its value, nor is its value ever displayed to them..." (Codd E:

Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems* 4(4):397 - 434, 1979).

Such schemas last about a year in actual use in an organization and then become unmanageable. To make this more concrete, let's model "vehicles," and the subclasses will be "automobiles," "SUVs," and so forth in a table like this:

```
CREATE TABLE VehicleClass
(id INTEGER NOT NULL AUTO_INCREMENT, -- not standard SQL
 attribute VARCHAR(255) NOT NULL,
 value VARCHAR(255) NOT NULL,
 subclass VARCHAR(255) NOT NULL,
..);
```

You will see this design referred to as an "entity-attribute-value" model in some of the literature. All the columns tend to be declared as the same long VARCHAR(n) or NVARCHAR(n) for a large value of (n), so that they can support strings that contain any numeric value, any temporal value, or any string that might hold the value of the entity's attribute. This now gives you overhead and possible errors of perpetual datatype conversions. You need to be sure that everyone uses the same formats for all the datatypes. Just think of all the ways that people enter date and time information and you have a rough idea of how bad this is going to be.

To find an entity you must assemble it from the pieces in the class table. Because some members of a class might not have exactly the same attributes as other members, you will tend to use a lot of expensive self OUTER JOINs in the queries.

Any typographical error becomes a new attribute. Consider adding a color attribute to the data model for a class of objects. The American programmer types in "color," the British programmer types in "colour," and the guy who is in a hurry types in "cloor." Nobody dares remove any of the attributes, even if they can find them all, because those attributes might belong to someone else's object.

Perhaps even worse, the names of such columns tend to become attempts to pass along the class hierarchy, physical storage, and usage information. The "color" attribute might be put into a table with column names, such as "color_code_id," "color_code_id_value," or worse. Likewise, you will see "i_color_code" if the code is an INTEGER. A data dictionary becomes almost impossible. (For details on how to name a data element, consult the ISO-11179.6 Metadata Standards naming conventions. The draft standards are at

http://pueblo.lbl.gov/~olken/X3L8/drafts/draft.docs.html, but there are many websites with information.) The use of NVARCHAR(n) has all these problems and the possibility that an entire Buddhist sutra in Chinese can be inserted as the value of an attribute.

It is extremely difficult to put constraints on such tables. Just consider the simple requirement that an employee be over 18 years of age. The birthdate and hire date of each employee has to be found, converted from VARCHAR(n) to a temporal datatype, the math performed, and the candidate rejected with a useful error code. You then need to decide what to do if one or both of those attributes are missing.

In short, you are using a high-level tool to try to build an OO database from the ground up, and it is an insane waste of time and resources. Does this mean that the idea of classes and relationships have no place in an SQL database? No, but they need to be implemented properly. There are some OO extensions in the SQL-99 Standard, but they are still not common in products and might not match the OO host language you are using.

In class hierarchies we are looking for sets of entities that are defined by common attributes, and then within that set we look for subsets with unique attributes. For example, personnel within a company all have job titles, tax identification numbers, and salaries. Within the personnel set the subset of salespeople also have a commission, the subset of executives also have stock options, and so forth.

The idea is to move from the general to the particular. This lets you handle the sets of entities at the appropriate level, based on the shared common attributes at that level.

10.1 Types of Hierarchies

A generalization hierarchy can either be overlapping or disjoint. In an overlapping hierarchy an entity can be a member of several subclasses. For example, people at a university could be broken into three subclasses: faculty, staff, and students. However, there is nothing to prevent the same person from belonging to two or more of these subclasses. A student could be on staff as part of a co-op program, a professor can take a class as a student, and so forth.

In a disjoint hierarchy an entity can be in one (and only one) subclass. For example, students at a university could be broken into three subclasses: foreign, in–state, and out-of-state students.

For the OO-minded reader, disjoint hierarchies are rather like single-inheritance type hierarchies, whereas overlapping hierarchies are like multiple-inheritance type hierarchies.

10.2 DDL Constraints

This is a nice set of definitions, but how do we code it in SQL? Here the hierarchy is not in one table, but is in the relationships among several tables.

10.2.1 Uniqueness Constraints

One of the basic tricks in SQL is representing a one-to-many relationship by creating a third table that references the two tables involved by their primary keys. This third table has quite a few popular names, such as "junction table" or "join table," but I know that it is a relationship. People tell you this, then leave you on your own to figure out the rest. For example, here are two tables:

```
CREATE TABLE Parents
(parent_name VARCHAR(30) NOT NULL PRIMARY KEY
... );

CREATE TABLE Children
(child_name VARCHAR(30) NOT NULL PRIMARY KEY,
... );

CREATE TABLE Families - wrong!
(parent_name VARCHAR(30) NOT NULL
         REFERENCES Parents (parent_name),
child_name VARCHAR(30) NOT NULL,
         REFERENCES Children (child_name));
```

"Families" does not have its own key; therefore I can have redundant duplicate rows. This mistake is easy to make. What is worse is that too often new programmers will try to correct the error by adding a key column to the table, often with some kind of proprietary auto-numbering feature. This actually makes the problem worse because the redundant duplicates can hide behind the auto-number and look like they are different.

There is a natural key in the form of PRIMARY KEY (parent_name, child_name), and it needs to be enforced. However, the only restriction the "Families" constraint gives us is that each (parent_name, child_name) pair appears only once. Every parent can be paired with every child, which is not what we wanted. Now, I want to make a rule that parents can have as many children as they want, but the children have to stick to one parent.

The way I do this is to use a NOT NULL UNIQUE constraint on the child_name column, which makes it a key. It's a simple key because it is only

one column, but it is also a nested key because it appears as a subset of the compound PRIMARY KEY.

"Families" is a proper table, without duplicated (parent_name, child_name) pairs, but it also enforces the condition that a child has a unique parent.

```
CREATE TABLE Families
(parent_name VARCHAR(30) NOT NULL
         REFERENCES Parents (parent_name),
child_name VARCHAR(30) NOT NULL UNIQUE, -- nested key
         REFERENCES Children (child_name),
PRIMARY KEY (parent_name, child_name) ); -- compound key
```

Notice that (parent_name, child_name) is actually a super-key because child_name is a key. You usually should avoid such redundancies, but SQL can only reference columns in UNIQUE() and PRIMARY KEY() constraints in the referencing table; therefore let me leave the code this way to help explain the purpose of the table.

Generalizing this schema is a bit complicated. Let's add a pet to the family and say that a pet belongs to one (and only one) child, but kids can have several pets. Another rule is that orphans cannot have pets. If orphans were allowed to have pets, then we would model the parent-children relationship with one table (Families) and model the child-pets' relationship with a second table. They would be distinct relationships, described by separate relationship tables.

Clearly I need to start with a pets' table.

```
CREATE TABLE Pets
(pet_name VARCHAR(30) NOT NULL PRIMARY KEY,
... );
```

My primary key is the full length of the type hierarchy, and the lowest subclass has to be unique.

```
CREATE TABLE Families -- wrong!
(parent_name VARCHAR(30) NOT NULL
         REFERENCES Parents (parent_name),
child_name VARCHAR(30) NOT NULL,
         REFERENCES Children (child_name),
pet_name VARCHAR(30) NOT NULL
         REFERENCES Pets(pet_name),
PRIMARY KEY (parent_name, child_name, pet_name));
```

But this has a serious problem. Consider the following data:

```
('Daddy', 'Billy', 'Rover')
('George', 'Billy', 'Rover')
('George', 'Billy', 'Fluffy')
```

We don't have a constraint to keep Billy from having two different parents, which leads to duplicates of 'Rover' in the table. Let's try adding some UNIQUE constraints.

```
CREATE TABLE Families -- wrong but better!
(parent_name VARCHAR(30) NOT NULL
         REFERENCES Parents (parent_name),
child_name VARCHAR(30) NOT NULL,
         REFERENCES Children(child_name),
pet_name VARCHAR(30) NOT NULL UNIQUE
         REFERENCES Pets(pet_name),
PRIMARY KEY (parent_name, child_name, pet_name));
```

With this table you have solved only part of the problem. I can get around this set of constraints by changing my table to:

```
('Daddy', 'Billy', 'Rover')
('George', 'Billy', 'Fluffy')
```

Billy still has two parents. We cannot use a UNIQUE (parent_name, child_name) constraint because this would not allow the child to have more than one pet. Change 'George' to 'Daddy' to see what I mean. Likewise, a UNIQUE (child_name, pet_name) constraint is redundant because the pet_name is unique. We are hitting the limits of Standard SQL uniqueness constraints.

One way around this is with a table level CHECK() constraint or a CREATE ASSERTION statement, thus

```
CREATE ASSERTION Only_One_Parent_per_Kid
CHECK (NOT EXISTS
        (SELECT *
           FROM Family AS F1
         GROUP BY child_name
        HAVING COUNT (parent_name) > 1));
```

The logical question at this point is, why not use this type of constraint to enforce the "child and pet" rule, thus

```
CREATE ASSERTION Only_One_Kid_per_Pet
CHECK (NOT EXISTS
        (SELECT *
          FROM Family AS F1
         GROUP BY pet_name
        HAVING COUNT (child_name) > 1));
```

These table level CHECK() constraints obviously generalize up the hierarchy. However, they have to be tested every time the table changes, so they can be quite expensive to execute, they do not improve access to the data, and they are not widely implemented yet. You would have to use a TRIGGER in most SQL products.

10.2.2 Disjoint Hierarchies

A simple way to enforce a disjoint hierarchy is with a matrix design. The relationship is stored in a table that connects each parent node to their proper child.

```
CREATE TABLE StudentTypes
(student_id INTEGER NOT NULL PRIMARY KEY
       REFERENCES Students (student_id)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
in_state INTEGER DEFAULT 0 NOT NULL
      CHECK (in_state IN (0, 1)),
out_of_state INTEGER DEFAULT 0 NOT NULL
        CHECK (out_of_state IN (0, 1)),
"foreign" INTEGER DEFAULT 0 NOT NULL
      CHECK ("foreign" IN (0, 1)),
CHECK ((in_state + out_of_state + "foreign") = 1));
```

To get to the particular attributes that belong to each subclass, you will need a table for that subclass. For example:

```
CREATE TABLE OutOfStateStudents
(student_id INTEGER NOT NULL PRIMARY KEY
            REFERENCES StudentTypes (student_id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
state CHAR(2) NOT NULL, -- USPS standard codes
... );
```

```
CREATE TABLE ForeignStudents
(student_id INTEGER NOT NULL PRIMARY KEY
            REFERENCES StudentTypes (student_id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
country_code CHAR(3) NOT NULL, -- ISO standard codes
... );

CREATE TABLE InStateStudents
(student_id INTEGER NOT NULL PRIMARY KEY
            REFERENCES StudentTypes (student_id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
county_code INTEGER NOT NULL, -- ANSI standard codes
high_school_district INTEGER NOT NULL,
... );
```

A more complex set of relationships among the subclass can also be enforced by making the CHECK() constraint more complex. The constant in the StudentTypes table can be changed from 1 to (n), the equality can be replaced with a less than, and so forth.

```
CHECK ((subclass_1 + subclass_2 + .. + subclass_n) <= k)
```

Another trick is to use powers of 2 so that each combination has a unique total. You can also use elaborate CASE expressions with a lot of business rules embedded in them.

Another version of the same approach uses a two-part key in the subclass tables in which one column is a constant that tells you what the table contains. Let's use an abbreviation code for 'in state,' 'out of state,' and 'foreign' students.

```
CREATE TABLE StudentTypes
(student_id INTEGER NOT NULL PRIMARY KEY
            REFERENCES Students (student_id)
            ON UPDATE CASCADE
            ON DELETE CASCADE,
residence_type CHAR(3) DEFAULT 'ins' NOT NULL
        CHECK (residence_type IN ('ins', 'out', 'for')));
```

Notice that if the key had been (student_id, residence_type), a student could appear in more than one subclass and we could add check constraints to

enforce various combinations of those subclasses. To get to the particular attributes that belong to each subclass, you will need a table for that subclass. For example:

```
CREATE TABLE OutOfStateStudents
(student_id INTEGER NOT NULL PRIMARY KEY
residence_type CHAR(3) DEFAULT 'out' NOT NULL
        CHECK (residence_type = 'out'),
FOREIGN KEY (student_id, residence_type)
REFERENCES StudentTypes (student_id, residence_type)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
 state CHAR(2) NOT NULL, -- USPS standard codes
 ...,
PRIMARY KEY (student_id, residence_type));

CREATE TABLE ForeignStudents
(student_id INTEGER NOT NULL
residence_type CHAR(3) NOT NULL
        CHECK (residence_type 'for'),
FOREIGN KEY (student_id, residence_type)
REFERENCES StudentTypes (student_id, residence_type)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
country_code CHAR(3) NOT NULL, -- ISO standard codes
...,
PRIMARY KEY (student_id, residence_type));

CREATE TABLE InStateStudents
(student_id INTEGER NOT NULL
residence_type CHAR(3) NOT NULL
        CHECK (residence_type 'ins'),
FOREIGN KEY (student_id, residence_type)
REFERENCES StudentTypes (student_id, residence_type)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
county_code INTEGER NOT NULL, -- ANSI standard codes
high_school_district INTEGER NOT NULL,
...,
PRIMARY KEY (student_id, residence_type));
```

The Declarative Referential Integrity (DRI) actions will enforce the class membership rules for us, but at the cost of redundant columns.

10.2.3 Representing 1:1, 1:m, and n:m Relationships

One of the basic tricks in SQL is representing a one-to-one or many-to-many relationship with a table that references the two (or more) entity tables involved by their primary keys. This third table has several popular names, such as "junction table" or "join table," but we know that it is a relationship. This type of table needs to have constraints to assure that the relationships work properly.

For example, I've given you two tables:

```
CREATE TABLE Boys
(boy_name VARCHAR(30) NOT NULL PRIMARY KEY
...);

CREATE TABLE Girls
(girl_name VARCHAR(30) NOT NULL PRIMARY KEY,
... );
```

I know that using names for a key is a bad practice, but it will make my examples easier to read. There are a lot of different relationships that we can make between these two tables. If you don't believe me, just watch your favorite television talk show. The simplest relationship table looks like this:

```
CREATE TABLE Pairs
(boy_name INTEGER NOT NULL
       REFERENCES Boys (boy_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
girl_name INTEGER NOT NULL,
       REFERENCES Girls (girl_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE);
```

The pairs table allows us to insert rows like this:

```
('Joe Celko', 'Marie Antoinette')
('Joe Celko', 'Cleopatra')
('Marc Anthony', 'Cleopatra')
('Joe Celko', 'Marie Antoinette')
```

Oops! I am shown twice with 'Marie Antoinette' because the pairs table does not have its own key. This is an easy mistake to make, but fixing it so that you enforce the proper rules is not obvious to a beginner.

```
CREATE TABLE Orgy
(boy_name INTEGER NOT NULL
       REFERENCES Boys (boy_name)
       ON DELETE CASCADE
       ON UPDATE CASCADE,
girl_name INTEGER NOT NULL,
       REFERENCES Girls (girl_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
PRIMARY KEY (boy_name, girl_name)); -- compound key
```

The orgy table gets rid of the duplicated rows and makes this a proper table. The primary key for the table is made up of two or more columns and is called a compound key because of that fact.

```
('Joe Celko', 'Marie Antoinette')
('Joe Celko', 'Cleopatra')
('Marc Anthony', 'Cleopatra')
```

The only restriction on the pairs is that they appear only once. Every boy can be paired with every girl, much to the dismay of the moral majority. I think I want to make a rule that guys can have as many gals as they want, but the gals have to stick to one guy.

The way I do this is to use a NOT NULL UNIQUE constraint on the girl_name column, which makes it a key. It is a simple key because it is only one column, but it is also a nested key because it appears as a subset of the compound PRIMARY KEY.

```
CREATE TABLE Polygany
(boy_name INTEGER NOT NULL
       REFERENCES Boys (boy_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
girl_name INTEGER NOT NULL UNIQUE, -- nested key
       REFERENCES Girls (girl_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
PRIMARY KEY (boy_name, girl_name)); -- compound key
```

The polygany is a proper table, without duplicated rows, but it also enforces the condition that I get to play around with one or more ladies, thus:

```
('Joe Celko', 'Marie Antoinette')
('Joe Celko', 'Cleopatra')
```

Of course, the ladies might want to go the other way and keep company with a series of men.

```
CREATE TABLE Polyandry
(boy_name INTEGER NOT NULL UNIQUE -- nested key
       REFERENCES Boys (boy_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
girl_name INTEGER NOT NULL,
       REFERENCES Girls (girl_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
PRIMARY KEY (boy_name, girl_name) ); -- compound key
```

The polyandry table would permit these rows from our original set:

```
('Joe Celko', 'Cleopatra')
('Marc Anthony', 'Cleopatra')
```

The moral majority is pretty upset about this scandal and would love for us to stop running around and settle down into nice, stable marriages.

```
CREATE TABLE Marriage
(boy_name INTEGER NOT NULL UNIQUE -- nested key
       REFERENCES Boys (boy_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
girl_name INTEGER NOT NULL UNIQUE -- nested key,
       REFERENCES Girls (girl_name)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
PRIMARY KEY (boy_name, girl_name) ); -- compound key
```

The marriage table allows us to insert these rows from the original set:

```
('Joe Celko', 'Marie Antoinette')
('Marc Anthony', 'Cleopatra')
```

Think about this table for a minute: the PRIMARY KEY is now redundant. If each boy appears only once in the table and each girl appears only once in the table, then each (boy_name, girl_name) pair can appear only once.

From a theoretical viewpoint I could drop the compound key and make either boy_name or girl_name the new primary key, or I could just leave them as candidate keys. However, SQL products and theory do not always match. Many products make the assumption that the PRIMARY KEY is, in some way, special in the data model and will be the way that they should access the table most of the time.

In fairness, making special provision for the primary key is not a bad assumption because the REFERENCES clause uses the PRIMARY KEY of the referenced table as a default. In many SQL products this can also give you a covering index for the query optimizer.

This page intentionally left blank

Hierarchical Encoding Schemes

A HIERARCHY IS A useful concept for classifying data, as well as retrieving it. The encoding schemes used to represent data are often hierarchical. Tree structures are a natural way to model encoding schemes that have a natural hierarchy. They organize the data for searching and reporting along that natural hierarchy and make it very easy for a human being to understand. But, what do you use for this natural organizational principle? Physical, temporal, or procedural options often exist, but many hierarchical encoding schemes are more circumstantial, traditional, and just plain arbitrary.

11.1 ZIP codes

The most common example of a hierarchical encoding scheme is the ZIP code, which partitions the United States geographically. Each digit, as you read from left to right, further isolates the location of the address first by postal region, then by state, then by city, and finally, by the post office that has to make the delivery. For example, given the ZIP code 30310, we know that the 30000 to 39999 range is in the southeastern United States. Within the southeastern codes we know that the 30000 to 30399 range is Georgia and that 30300 to 30399 is metropolitan Atlanta. Finally, the whole code, 30310, identifies substation 'A' in the West End section of the city. The ZIP code can be parsed by reading it from left to right, reading first one digit, then two, and then the last two digits.

There are many websites that look up the cities in the United States by their ZIP codes, compute the distance between two ZIP codes, and so forth (http://zip.langenberg.com/). Each ZIP code has a preferred city, but a suburb or sister town might fall under the same code if they are small enough or are served by the same post office. Likewise, an address in a town that goes over a state border might have a ZIP code that actually belongs to the other state. In short, it is not a perfect locator for (city, state) combinations, but it is close enough for making contacts by mail or physical location.

In 1983 the U.S. Postal Service began using an expanded ZIP Code, the ZIP + 4, which consists of the original five-digit ZIP Code plus a four-digit add-on code. The four-digit add-on number identifies a geographic segment within the five-digit delivery area, such as a city block, office building, individual high-volume receiver of mail, or any other physical unit that would aid sorting and delivery. The ZIP + 4 codes are not required for first class mail, but must be used with certain classes of bulk mail to aid machine presorting.

11.2 Dewey Decimal Classification

Melville Louis Kossuth Dewey (1851-1931) had two manias in his life—spelling reform and libraries.

As an aside, spelling reform was a hot topic in the United States at that time and that's when most of the differences between British and American English were established. Dewey even used "reformed spelling" in several editions of the Dewey Decimal Classiciation (DDC). He changed his name to "Melvil Dui," thereby dropping his middle names, but finally changed the family name back to the original spelling.

He invented the DDC when he was a 21-year-old student assistant in the Amherst College (Amherst, MA) library. What is hard for us to imagine is that before the DDC every library made up its own classification system without recourse to any standard model. It sounds a lot like IT shops today, doesn't it?

Dewey helped establish the American Library Association (ALA) in 1876 while he was the librarian of Columbia College (now Columbia University) in New York; he founded the first library school in 1887 and raised librarianship to a profession.

The DDC was undergoing its 21st revision in 2002. Hard copy and electronic formats can be ordered from the OCLC (Online Computer Library Center, Inc.):

OCLC
6565 Frantz Road
Dublin, OH 43017-3395
USA
email: dewey@oclc.org

The 500-number series covers "Natural Sciences & Mathematics"; within that the 510s cover "Mathematics"; finally, 512 deals with "Algebra & Number Theory," in particular. The scheme could be carried further, with decimal fractions for various kinds of algebra.

11.3 Strength and Weaknesses

Hierarchical encoding schemes are great for large data domains that have a natural hierarchy, but there can be problems in designing these schemes. First, the tree structure does not have to be neatly balanced, so some categories may need more codes than others and hence more breakdowns. Eastern and ancient religions are shortchanged in the DDC, reflecting a prejudice toward Christian writings. Asian religions were pushed into a very small set of codes. Today the Library of Congress has more books on Buddhist thought than on any other religion on Earth. Second, you might not have made the right choices as to where to place certain values in the tree. For example, in the DDC, books on logic are encoded as 160, in the philosophy section, and not under the 510s, mathematics (Box 11.1). In the nineteenth century, there was no mathematical logic. Today there is no philosophical logic. Dewey was simply following the conventions of his day. Like today's programmers, he found that the system specifications changed while he was working.

Why this particular breakdown of human knowledge? Well, why not? Besides, it could be much worse. Let me give you a quote from the essay, "The Analytical Language of John Wilkins" by Jorge Luis Borges: "These ambiguities, redundancies, and deficiencies recall those attributed by Dr. Franz Kuhn to a certain Chinese encyclopedia entitled Celestial Emporium of Benevolent Knowledge. On those remote pages it is written that animals are divided into (a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance."

Box 11.1 Dewey Decimal Table Search for 'Logic'

The Hundreds Level (Overview)

000 Generalities
100 Philosophy & psychology
200 Religion
300 Social sciences
400 Language
500 Natural sciences & mathematics
600 Technology (applied sciences)
700 The arts
800 Literature & rhetoric
900 Geography & history

The Tens Level

160 Logic

The Units Level

161 Induction
162 Deduction
163 Not assigned or no longer used
164 Not assigned or no longer used
165 Fallacies & sources of error
166 Syllogisms
167 Hypotheses
168 Argument & persuasion
169 Analogy

11.4 Shop Categories

In the retail industry, stores will often set up their own shop categories to classify their merchandise. For example, if you go to a larger bookstore you will see a separate juvenile section, sections for romance novels, and so forth. Within these sections you might find books grouped alphabetically by authors or by further subclassifications.

These shop category tables are hard for beginning SQL programmers to design because the designers have a hard time conceptually divorcing the categories from the merchandise. This will be easier to see with an example, which was taken from an actual posting on a Usenet newsgroup.

First, set up a simplified inventory table, which uses the UPC code to identify the merchandise.

```
CREATE TABLE Inventory
(upc DECIMAL(10,0) NOT NULL,
product_name VARCHAR(200) NOT NULL,
category_id INTEGER NOT NULL,
quantity_on_hand INTEGER NOT NULL);
```

Each product has a category, but here is what the first attempt at a categories table will look like:

```
CREATE TABLE Categories
(category_id INTEGER NOT NULL,
category_parent INTEGER NULL, -- null means root
category_name VARCHAR(200) NOT NULL,
category_count INTEGER NULL DEFAULT (0))
```

As you can see, each category has a category_parent that symbolize the higher category in a hierarchy. The original poster wanted to know if he could do "some kind of a loop" for each category_id and tally the quantity on hand into the category_count, with the proper nesting of the categories beneath. There are several design problems in this schema. A better approach is shown in the next table. First, check to see that the category_id is within the boundaries of the category classification system.

```
CREATE TABLE Inventory
(upc DECIMAL(10,0) NOT NULL PRIMARY KEY,
 product_name CHAR(20) DEFAULT 'unknown' NOT NULL,
 category_id INTEGER NOT NULL
           CHECK (category_id BETWEEN 000 AND 999),
 quantity_on_hand INTEGER NOT NULL);
```

However, the real problem is that the categories table is wrong. Using the basic idea of the nested sets model, we can set up ranges, such as Dewey Decimal Classification system, and add more constraints to the table:

```
CREATE TABLE Categories
(category_name CHAR(20) DEFAULT 'unknown' NOT NULL
            PRIMARY KEY,
category_low INTEGER NOT NULL UNIQUE,
category_high INTEGER NOT NULL UNIQUE,
```

```
CHECK (category_low <= category_high));

INSERT INTO Categories VALUES ('Printers (all)', 500, 599);
INSERT INTO Categories VALUES ('InkJet Printers', 510, 519);
INSERT INTO Categories VALUES ('Laser Printers', 520, 529);
```

Instead of doing a loop and trying to keep the total in a column in the categories table, use this VIEW, which will always be right, always up-to-date, and will show all categories.

```
CREATE VIEW CategoryReport (category_name, total_qty)
AS SELECT C1.category_name, COALESCE(SUM(quantity_on_hand), 0)
     FROM Categories AS C1
          LEFT OUTER JOIN
          Inventory AS P1
          ON P1.category_id
             BETWEEN C1.category_low AND C1.category_high
    GROUP BY C1.category_name;
```

If you wanted the category hierarchy to end with an actual inventory entity, you can enforce this with a declarative referential integrity constraint. In the case of a bookstore the categories would probably not go down to individual titles, but a retail computer store would like to go to the make and model of their equipment, with an entry such as:

```
INSERT INTO Categories
VALUES ('Fonebone X-7 Laser Printer', 521, 521);
```

You then use two REFERENCES clauses on the same column to make sure that each inventory item is represented in the categories table, thus:

```
CREATE TABLE Inventory
(product_name VARCHAR(200) NOT NULL,
 category_id INTEGER NOT NULL UNIQUE
             REFERENCES Categories(range_start)
             ON UPDATE CASCADE
             ON DELETE CASCADE,
             REFERENCES Categories(range_end)
             ON UPDATE CASCADE
             ON DELETE CASCADE,
 quantity_on_hand INTEGER NOT NULL);
```

A good rule of thumb is that you need to use a range of numbers that is bigger than what you need now. Data has a way of growing.

11.5 Statistical Tools for Decision Trees

You can buy statistical tools that look at raw data and cluster it by attribute values into a hierarchy based on that data. These are generally used for data mining, so I will only mention them in passing and give a simple example.

Using a sample database from KnowledgeSeeker (Angoss Software), you start with a series of records about the lifestyles of people and their blood pressure: How much do they drink?; How much do they smoke?; exercise?; What foods do they eat?; and so forth. The KnowledgeSeeker engine takes the data and produces a tree diagram and a set of rules for predicting blood pressure (the dependent variable) from the other information (independent variables).

At the first level of the tree we find that age is the most important factor, and we have three subgroups. Within the younger age group (32 - 50 years) you need to stop heavy drinking; within the middle-aged age group (51 - 62 years) you need to stop smoking; and within the oldest age group (63 to 72 years), if you have survived a lifetime of smoking and drinking, you need to watch your diet now. Using this information and a questionnaire, I can predict the likelihood of a new patient having high blood pressure.

However, as my sample size changes or I add more attributes (e.g., family medical history in the example), my tree might need to be recomputed and decisions reevaluated based on the more current and/or complete data available to me.

This page intentionally left blank

Hierarchical Database Systems (IMS)

I AM GOING TO assume that most of the readers of this book have only worked with SQL. If you have heard of a hierarchical database system, it was mentioned in a database course in college and then forgotten. In some ways this is too bad. It helps to know how the earlier tools worked so that you can see how the new tools evolve from the old ones.

The following material is taken from a series on IMS that appeared in www.dbazine.com. This is not going to make you an IMS programmer, but should help give you an overview. Why IMS? It is the most important prerelational technology that is still in wide use today. In fact, there is a good chance that IMS databases still hold more data than SQL databases.

12.1 Types of Databases

The classic types of database structures are network, relational, and hierarchical. The network and hierarchical models are called network or "navigational" databases because the mental model of data access is that of a reader moving along paths to pick up the data. In fact, when Charles Bachman received the ACM Turing Award, this is how he described it. (ACM Turing Award speech 1973, "The Programmer as Navigator" by Charles Bachman; http://www.ischool.washington.edu/tabrooks/100/Documents/Bachman/ProgrammerNavigator.pdf)

IMS was not the only navigational database, just the most popular. TOTAL from CinCom was based on a Master record that had pointer chains to one or

more sets of slave records. Later, IDMS and other products generalized this navigational model.

CODASYL, the committee that defined COBOL, came up with a standard for the navigational model. Finally, the ANSI X3H2 Database Standards Committee took the CODASYL model, formalized it a bit, and produced the NDL language specification. However, at that point SQL had become the main work of the ANSI X3H2 Database Standards Committee and nobody really cared about NDL, and the standard simply expired.

Because this is a book on hierarchies and relational databases, I am going to ignore the network model on the assumption that it is too old and the products too varied to be of interest. I am also going to ignore object-oriented and other "postrelational" databases on the assumption that they are too young, too varied, and too uncommon to be of interest.

IMS from IBM is the one hierarchical database management system still in wide use today. It is stable, well-defined, scalable, and very fast for what it does. The IMS software environment can be divided into five main parts: (1) database, (2) data language I (DL/I), (3) DL/I control blocks, (4) data communications component (IMS TM), and (5) application programs.

Figure 12.1 is a diagram of the relationships of the IMS components. We will discuss some of these components in more detail, but not in great detail.

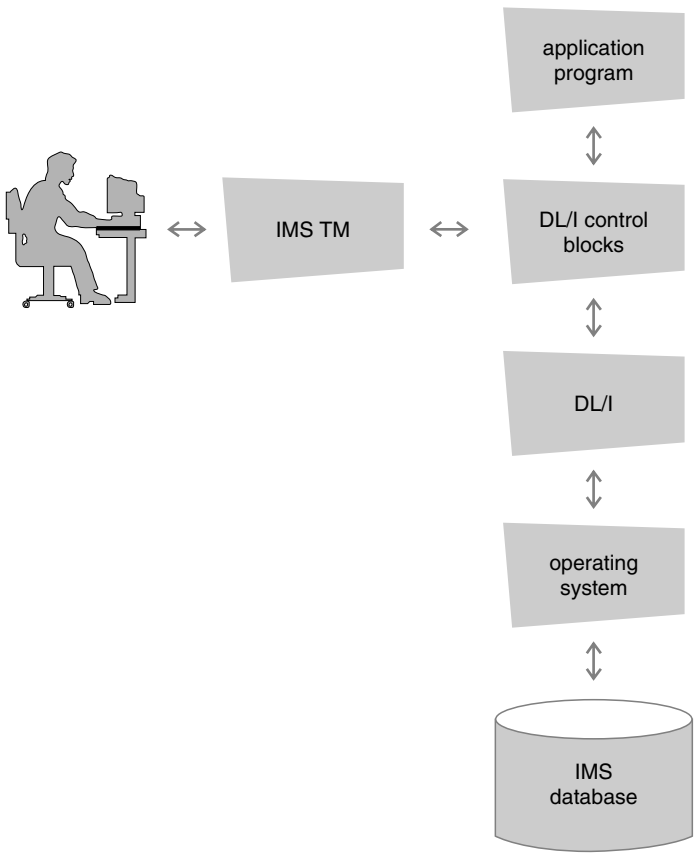
12.2 Database History

Before the development of DBMSs, data was stored in individual files. With this system each file was stored in a separate data set in sequential or indexed format. To retrieve data from the file, an application had to open the file and read through it to the location of the desired data. If the data was scattered through a large number of files, data access required a lot of opening and closing of files, creating additional I/O and processing overhead.

To reduce the number of files accessed by an application, programmers often stored the same data in many files. This practice created redundant data and the related problems of ensuring update consistency across multiple files. To ensure data consistency, special cross-file update programs had to be scheduled following the original file update.

The concept of a database system resolved many data integrity and data duplication issues encountered in a file system. A properly designed database stores the data only once in one place and makes it available to all application programs and users. At the same time databases provide security by limiting access to data. The user's ability to read, write, update, insert, or delete data

Fig. 12.1



can be restricted. Data can also be backed up and recovered more easily in a single database than in a collection of flat files.

Database structures offer multiple strategies for data retrieval. Application programs can retrieve data sequentially or (with certain access methods) go directly to the desired data, reducing I/O and speeding data retrieval. Finally, an update performed on part of the database is immediately available to other applications. Because the data exists in only one place, data integrity is more easily ensured.

The IMS database management system as it exists today represents the evolution of the hierarchical database over many years of development and improvement. IMS is in use at a large number of business and government installations throughout the world. IMS is recognized for providing excellent

performance for a wide variety of applications and for performing well with databases of moderate to very large volumes of data and transactions.

12.2.1 DL/I

Because they are implemented and accessed through use of the Data Language I, IMS databases are sometimes referred to as DL/I databases. DL/I is a command-level language, not a database management system. DL/I is used in batch and online programs to access data stored in databases.

Application programs use DL/I calls to request data. DL/I then uses system access methods, such as Virtual Storage Access Method (VSAM), to handle the physical transfer of data to and from the database.

IMS databases are often referred to by the access method they are designed for, such as HDAM, PHDAM, HISAM, HIDAM, and PHIDAM. These are all IBM terms from their mainframe database products and I will not discuss them here.

IMS makes provisions for nine types of access methods, and you can design a database for any one of them. On the other hand, SQL programmers are generally isolated from the access methods that their database engine uses. We will not worry about the details of the access methods that are called at this level.

12.2.2 Control Blocks

When you create an IMS database, you must define the database structure and how the data can be accessed and used by application programs. These specifications are defined within the parameters provided in two control blocks, also called DL/I control blocks: (1) Database description (DBD) and (2) Program specification block (PSB).

In general the DBD describes the physical structure of the database, and the PSB describes the database as it will be seen by a particular application program. The PSB tells the application which parts of the database it can access and the functions it can perform on the data. Information from the DBD and PSB is merged into a third control block, the application control block (ACB). The ACB is required for online processing but is optional for batch processing.

12.2.3 Data Communications

The IMS Transaction Manager (IMS TM) is a separate set of licensed programs that provide access to the database in an online, real-time environment.

Without the TM component you would be able to process data in the IMS database in a batch mode only.

12.2.4 Application Programs

The data in a database is of no practical use to you if it sits in the database untouched. Its value comes in its use by application programs in the performance of business or organizational functions. With IMS databases, application programs use DL/I calls embedded in the host language to access the database. IMS supports batch and online application programs. IMS supports programs written in assembler, C, COBOL, PL/I, Pascal, and REXX.

12.2.5 Hierarchical Databases

In a hierarchical database, data is grouped in records, which are subdivided into a series of segments. Consider a Departmental database for a school in which a record consists of the segments Dept, Course, and Enroll. In a hierarchical database the structure of the database is designed to reflect logical dependencies—certain data is dependent on the existence of certain other data. Enrollment is dependent on the existence of a course, and in this case a course is dependent on the existence of a department to offer that course.

The terminology changes from the SQL world to the IMS world. IMS uses records and fields, and calls each hierarchy a database. In the SQL world a row and column are similar to record and field, but are much smarter and more general. In SQL a schema or database is a collection of related tables, which might map into several different IMS hierarchies in the same data model. In other words, an IMS database is more like a table in SQL.

12.2.6 Strengths and Weaknesses

In a hierarchical database the data relationships are defined by the storage structure. The rules for queries are highly structured. It is these fixed relationships that give IMS extremely fast access to data when compared to an SQL database, when the queries have not been highly optimized.

Hierarchical and relational systems have their strengths and weaknesses. The relational structure makes it relatively easy to code ad hoc queries. However, an SQL query often takes the engine through an entire table or series of tables to retrieve the data. This makes searches slower and more processing-intensive. In addition, because the row and column structure must be maintained throughout the database, an entry must be made under each

column for every row in every table, even if the entry is only a placeholder (i.e., NULL) entry.

With the hierarchical structure data requests or segment search arguments (SSAs) may be more complex to construct. Once written, however, they can be very efficient, allowing direct retrieval of the data requested. The result is an extremely fast database system that can handle huge volumes of data transactions and large numbers of simultaneous users. Likewise, there is no need to enter placeholders where data is not being stored. If a segment occurrence isn't needed, it isn't created or inserted.

In 25 words or less the trade-offs are the simplicity, portability, and flexibility of SQL versus the speed and storage savings of IMS. You tune an IMS database for one set of applications.

12.3 Sample Hierarchical Database

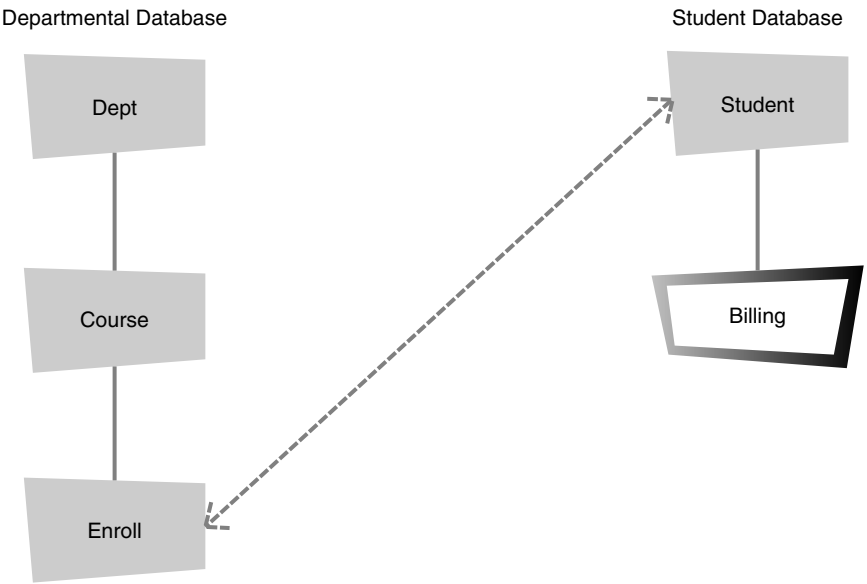
To illustrate how the hierarchical structure looks, we'll design two very simple databases to store information for the courses and students in a college. One database will store information on each department in the college, and the second will contain information on each college student. In a hierarchical database an attempt is made to group data in a one-to-many relationship.

An attempt is also made to design the database so that data that is logically dependent on other data is stored in segments that are hierarchically dependent on the data. For that reason we have designated Dept as the key, or root, segment for our record, because the other data would not exist without the existence of a department. We list each department only once. We provide data on each course in each department. We have a segment type Course, with an occurrence of that type of segment for each course in the department. Data on the course title, description, and instructor is stored as fields within the Course segment. Finally, we have added another segment type, Enroll, which will include the student IDs of the students enrolled in each course.

In Figure 12.2 we also created a second database called Student. This database contains information on all the students enrolled in the college. This database duplicates some of the data stored in the Enroll segment of the Departmental database. Later we will construct a larger database that eliminates the duplicated data. The design we choose for our database depends on a number of factors; in this case we will focus on which data we will need to access most frequently.

The two sample databases, Departmental and Student, are shown in Figure 12.2. The two databases are shown as they might be structured in relational form in three tables.

Fig. 12.2




```
CREATE Schema College

CREATE TABLE Courses
(course_nbr CHAR(9) NOT NULL PRIMARY KEY,
 course_title VARCHAR(20) NOT NULL,
 description VARCHAR(200) NOT NULL,
 dept_id CHAR(7) NOT NULL
       REFERENCES Departments (dept_id)
      ON UPDATE CASCADE);

CREATE TABLE Students
(student_id CHAR(9) NOT NULL PRIMARY KEY,
 student_name CHAR(35) NOT NULL,
 address CHAR(35) NOT NULL,
 major CHAR(10));

CREATE TABLE Departments
(dept_id CHAR(7) NOT NULL PRIMARY KEY,
 dept_name CHAR(15) NOT NULL,
 chairman CHAR(35) NOT NULL,
 budget_code CHAR(3) NOT NULL);
```

12.3.1 Departmental Database

The segments in the Departmental database are as follows:

Dept: Information on each department. This segment includes fields for the department ID (the key field), department name, chairman's name, number of faculty, and number of students registered in departmental courses.

Course: This segment includes fields for the course number (a unique identifier), course title, course description, and instructor's name.

Enroll: The students enrolled in the course. This segment includes fields for student ID (the key field), student name, and grade.

12.3.2 Student Database

The segments in the Student database are as follows:

Student: Student information. It includes fields for student ID (key field), student name, address, major, and courses completed.

Billing: Billing information for courses taken. It includes fields for semester, tuition due, tuition paid, and scholarship funds applied.

The dotted line between the root (Student) segment of the Student database and the Enroll segment of the Departmental database represents a logical relationship based on data residing in one segment and needed in the other.

12.3.3 Design Considerations

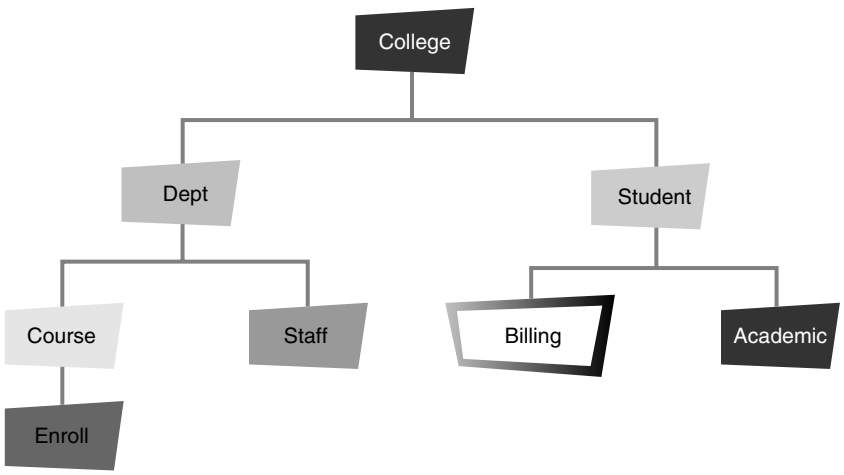
Before implementing a hierarchical structure for your database, you should analyze the end user's processing requirements, because they will determine how you structure the database. In particular, you must consider how the data elements are related and how they will be accessed.

For example, given parts and suppliers, the hierarchical structure could subordinate parts under suppliers for the accounts receivable department, or subordinate suppliers under parts for the order department.

12.3.4 Example Database Expanded

At this point we have learned enough about database design to expand our original example database. We decide that we can make better use of our college data by combining the Departmental and Student databases. Our new College database is shown in Figure 2.3.

Fig. 12.3



The following segments are in the expanded College database:

College: The root segment. One record will exist for each college in the university. The key field is the College ID, such as ARTS, ENGR, BUSADM, and FINEARTS.

Dept: Information on each department within the college. It includes fields for the department ID (the key field), department name, chairman's name, number of faculty, and number of students registered in departmental courses.

Course: Includes fields for the course number (the key field), course title, course description, and instructor's name.

Enroll: A list of students enrolled in the course. There are fields for student id (key field), student name, current grade, and number of absences.

Staff: A list of staff members, including professors, instructors, teaching assistants, and clerical personnel. The key field is employee number. There are fields for name, address, phone number, office number, and work schedule.

Student: Student information. It includes fields for student ID (key field), student name, address, major, and courses being taken currently.

Billing: Billing and payment information. It includes fields for billing date (key field), semester, amount billed, amount paid, scholarship funds applied, and scholarship funds available.

Academic: The key field is a combination of the year and the semester. Fields include grade point average per semester, cumulative GPA, and enough fields to list courses completed and grades per semester.

12.3.5 Data Relationships

The process of data normalization helps you break data into naturally associated groupings that can be stored collectively in segments in a hierarchical database. In designing your database break the individual data elements into groups, based on the processing functions they will serve. At the same time, group data based on inherent relationships between data elements.

For example, the College database (Figure 12.3) contains a segment called Student. Certain data is naturally associated with a student, such as student ID number, student name, address, and courses taken. Other data that we will want in our College database, such as a list of courses taught or administrative information on faculty members, would not work well in the Student segment.

Two important data-relationship concepts are one-to-many and many-to-many. In the College database there are many departments for each college (see Figure 12.3, which shows only one example), but only one college for each department. Likewise, many courses are taught by each department, but a specific course (in this case) can be offered by only one department.

The relationship between courses and students is many-to-many, as there are many students in any course, and each student will take several courses. Let's ignore the many-to-many relationship for now; this is the hardest relationship to model in a hierarchical database.

A one-to-many relationship is structured as a dependent relationship in a hierarchical database: the many are dependent on the one. Without a department there would be no courses taught, and without a college, there would be no departments.

Parent and child relationships are based solely on the relative positions of the segments in the hierarchy, and a segment can be a parent of other segments while serving as the child of a segment above it. In Figure 12.3 Enroll is a child of Course, and Course, although the parent of Enroll, is also the child of Dept.*

---

*Billing and Academic are both children of Student, which is a child of College. (Technically, all of the segments, except College, are dependents.)

When you have analyzed the data elements, grouped them into segments, selected a key field for each segment, and designed a database structure, you have completed most of your database design. You may find, however, that the design you have chosen does not work well for every application program. Some programs may need to access a segment by a field other than the one you have chosen as the key. Another application may need to associate segments that are located in two different databases or hierarchies. IMS has provided two very useful tools that you can use to resolve these data requirements: secondary indexes and logical relationships.

Secondary indexes let you create an index based on a field other than the root segment key field. That field can be used as if it were the key to access segments based on a data element other than the root key.

Logical relationships let you relate segments in separate hierarchies and, in effect, create a hierarchical structure that does not actually exist in storage. The logical structure can be processed as if it physically exists, allowing you to create logical hierarchies without creating physical ones.

12.3.6 Hierarchical Sequence

Because segments are accessed according to their sequence in the hierarchy, it is important to understand how the hierarchy is arranged. In IMS, segments are stored in a top-down, left-to-right sequence (Figure 12.4). The sequence flows from the top to the bottom of the leftmost path or leg. When the bottom of that path is reached, the sequence continues at the top of the next leg to the right.

Understanding the sequence of segments within a record is important to understanding movement and position within the hierarchy. Movement can be forward or backward and always follows the hierarchical sequence. Forward means from top to bottom, and backward means bottom to top. Position within the database means the current location at a specific segment. You are once more doing depth-first tree traversals, but with a slightly different terminology.

12.3.7 Hierarchical Data Paths

In Figure 12.4 the numbers inside the segments show the hierarchy as a search path would follow it. The numbers to the left of each segment show the segment types as they would be numbered by type, not occurrence; that is, there may be any number of occurrences of segment type 04, but there will be only one type of segment 04. The segment type is referred to as the segment code.

To retrieve a segment, count every occurrence of every segment type in the path and proceed through the hierarchy according to the rules of navigation:

1. top to bottom
2. front to back (counting twin segments)
3. left to right

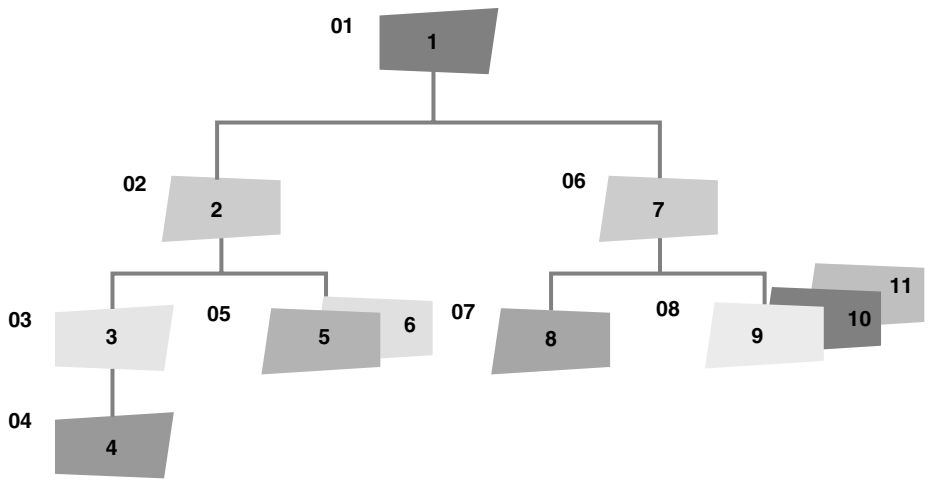
For example, if an application program issues a GET-UNIQUE (GU) call for segment 6 in Figure 12.4, the current position in the hierarchy is immediately following segment 06. If the program then issued a GET-NEXT (GN) call, IMS would return segment 07. There is also the GNP (Get Next within Parent) call, which explains itself.

As shown in Figure 12.4, the College database can be separated into four search paths. The first path includes segment types 01, 02, 03, and 04. The second path includes segment types 01, 02, and 05. The third path includes segment types 01, 06, and 07. The fourth path includes segment types 01, 06, and 08. The search path always starts at 01, which is the root segment.

12.3.8 Database Records

Whereas a database consists of one or more database records, a database record consists of one or more segments. In the College database a record consists of the root segment College and its dependent segments. It is possible to define a

Fig. 12.4



database record as only a root segment. A database can contain only the record structure defined for it, and a database record can contain only the types of segments defined for it.

The term *record* can also be used to refer to a data set record (or block), which is not the same thing as a database record. IMS uses standard data-system management methods to store its databases in data sets. The smallest entity of a data set is also referred to as a record (or block).

Two distinctions are important. A database record may be stored in several data set blocks. A block may contain several whole records or pieces of several records. In this chapter we try to distinguish between database record and data set record, in which the meaning may be ambiguous.

12.3.9 Segment Format

A segment is the smallest structure of the database in the sense that IMS cannot retrieve data in an amount less than a segment. Segments can be broken down into smaller increments called fields, which can be addressed individually by application programs.

A database record can contain a maximum of 255 types of segments. The number of segment occurrences of any type is limited only by the amount of space you allocate for the database. Segment types can be of fixed length or variable length. You must define the size of each segment type.

It is important to distinguish the difference between segment types and segment occurrences. Course is a type of segment defined in the DBD for the College database. There can be any number of occurrences for the Course segment type. Each occurrence of the Course segment type will be exactly as defined in the DBD. The only difference in occurrences of segment types is the data contained in them (and the length, if the segment is defined as variable length).

Segments have several different possible structures, but from a logical viewpoint, there is a prefix that has structural and control information for the IMS system, and 3 is the prefix for the actual data fields.

In the data portion you can define the following types of fields: a sequence field and the data fields.

Sequence (Key) Field: The sequence field is often referred to as the key field. It can be used to keep occurrences of a segment type in sequence under a common parent, based on the data or value entered in this field. A key field can be defined in the root segment of a HISAM, HDAM, or HIDAM database to give an application program direct access to a specific root segment. A key field can be used in HISAM and HIDAM databases to allow database records

to be retrieved sequentially. Key fields are used for logical relationships and secondary indexes.

The key field not only can contain data, but also can be used in special ways that help you organize your database. With the key field you can keep occurrences of a segment type in some kind of key sequence, which you design. For instance, in our example database you might want to store the student records in ascending sequence, based on student ID number. To do this you define the student ID field as a unique key field. IMS will store the records in ascending numerical order. You could also store them in alphabetical order by defining the name field as a unique key field. The following three factors of key fields are important to remember:

1. The data or value in the key field is called the key of the segment.
2. The key field can be defined as unique or non-unique.
3. You do not have to define a key field in every segment type.

Data: You define data fields to contain the actual data being stored in the database. (Remember that the sequence field is a data field.) Data fields, including sequence fields, can be defined to IMS for use by applications programs.

12.3.10 Segment Definitions

In IMS, segments are defined by the order in which they occur and by their relationship with other segments:

Root segment: The first, or highest, segment in the record. There can be only one root segment for each record. There can be many records in a database.

Dependent segment: All segments in a database record, except the root segment.

Parent segment: A segment that has one or more dependent segments beneath it in the hierarchy.

Child segment: A segment that is a dependent of another segment above it in the hierarchy.

Twin segment: A segment occurrence that exists with one or more segments of the same type under a single parent.

There are functions to edit, encrypt, or compress segments, which we will not consider here. The point is that you have a lot of control of the data at the physical level in IMS.

12.4 Summary

"Those who cannot remember the past are condemned to repeat it."—George Santayana

There were databases before SQL, and they were all based on a graph theory model. What SQL programmers do not like to admit is that less than 20% of all commercial information resides in SQL databases. The majority is still in simple files or older, navigational, nonrelational databases.

Even after the new tools have taken on their own characteristics to become a separate species, the mental models of the old systems still linger. The old patterns are repeated in the new technology.

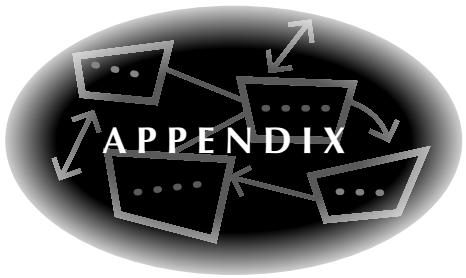
Even the early SQL products fell into this trap. For example, how many SQL programmers use IDENTITY or other auto-increment vendor extensions as keys on SQL tables today, unaware that they are imitating the sequence field (aka the "key field") from IMS?

This is not to say that a hierarchy is not a good way to organize data; it is! But you need to see the abstraction apart from any particular implementation. SQL is a declarative language, whereas DL/I is a collection of procedure calls inside a host language. The temptation is to continue to write SQL code in the same style as you wrote procedural code in COBOL, PL/I, or whatever host language you had.

The bad news is that you can use cursors to imitate sequential file routines. Roughly, the READ() command becomes an embedded FETCH statement, OPEN and CLOSE file commands map to OPEN and CLOSE CURSOR statements, and every file becomes a simple table without any constraints and a "record number" of some sort. The conversion of legacy code is almost effortless with such a mapping. In addition, it is also the worst way to program with a SQL database.

Hopefully this book will show you a few tricks that will let you write SQL as SQL and not fake a previous language in it.

This page intentionally left blank



Readings and Resources

General References

The website www.dbazine.com has a detailed three-part tutorial on IMS, from which material for Chapter 12 was brutally extracted and summarized.

The best source for IMS materials is at www.redbooks.ibm.com/, where you can download manuals directly from IBM.

Graph Theory

Here is a short list of introductory books on graph theory. None of them are specifically about trees, but they all have good chapters on that topic.

Berge, Claude. 2001. *Theory of Graphs*. Mineola: Dover Books. *ISBN: 0-486-41975-4*.

Chartrand, Gary. 1985. *Introductory Graph Theory*. Mineola: Dover Books. *ISBN: 0-486-24775-9. This book uses lots of real world examples and problems to introduce topics in graph theory. Very readable*.

Eve, Shimon. 1979. *Graph Algorithms*. Rockville: Computer Science Press. *ISBN 0914894-21-8. This book is more oriented toward programmers*.

Harary, Frank. 1972. *Graph Theory*. New York: Addison-Wesley. *ISBN 0-201-02787-9. Frank Harary is one of the leading mathematicians in graph theory. Therefore, this book is a bit more oriented toward proofs, but it is quite readable*.

McHugh, James A. 1990. *Algorithmic Graph Theory*. New York: Prentice Hall. *ISBN 0-13-23615-2. This book is more oriented toward programmers.*

Ore, Oystein (revised by Robin J. Wilson). 1990. *Graphs and Their Uses*. Washington, DC: American Mathematical Association. *ISBN 0-88358-635-2. This was one of a series of books from the AMA used for the introduction of high school students to advanced topics in mathematics.*

Trudeau, Richard J. 1994. *Introduction to Graph Theory*. Mineola: Dover Books. *ISBN: 0-486-67870-9.*

Trees operators in SQL

Date, Chris. 1986. "A Note on the Parts Explosion Problem." *Relational Database Selected Writings*. pp. 397-416. New York: Addison-Wesley. *ISBN 0-201-14196-5.*

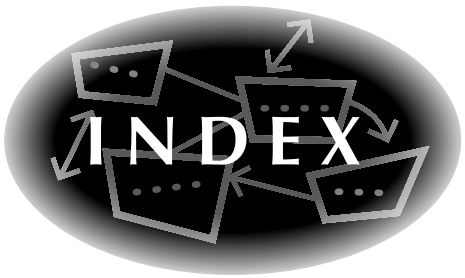
——— . 1986. "Why is it so Difficult to Provide a Relational Interface to IMS?" *Relational Database Selected Writings.* pp. 241-257. New York: Addison-Wesley. *ISBN 0-201-14196-5.*

Tillquist, John and Kuo, Feng-Yang. 1989. "An Approach to the Recursive Retrieval Problem in the Relational Database." *CACM*. 32(2): 239-245.

Bill of Materials in SQL

Blaha, M., Premerlini, W., Bender, A., Salemme, R., Kornfien, M., and Harkins, C. "Bill of Materials Configuration Generation." Sixth International Conference on Data Engineering. 1990 Feb 5-9. Los Angeles, CA.

Schmitz, Peter. 1981. "Using Ingres for Bill of Materials Problems." Relational Technology, Inc.



A

ACB. *See* Application control block
ACCESS, nested sets model in, 99
ACM Turing Award, 199
Adelson-Velskii, G. M., 144
Adjacency and depth model, 160–161
Adjacency and nested sets model, 159–160
Adjacency arrays, 6–7
Adjacency list model, 17
  converting, 90–92
  finding subordinates in, 56
  fixing, 22–25
  leveled, 31–32
  navigation in, 25–28
  nodes in, 28
  normalization and, 19–22
  procedural code and, 34
  simple, 17–19
  SQL databases and, 89
Adjacency lists
  for graphs, 6
  with self-references, 157–158
  subordinate, 158–159
Advanced Transact-SQL for SQL Server 2000 (Ben-Gan & Moreau), 35
Aggregate functions, 69
Aggregation
  based on self-joins, 27
  in hierarchy, 33–34
American Library Association (ALA), 192
Anomalies
  DELETE, 21
  INSERT, 20–21
  normalization of, 19
  structural, 22
  UPDATE, 19–20
ANSI X3H2 Database Standards Committee, 175
  CODASYL and, 200
Application control block (ACB), 202
Arrays
  for data structure, 5
  for graphs, 5
  in SQL-99, 7
  two-dimensional, 6
The Art of Computer Programming (Knuth), 45
Assembly language, 4–5
Auto-numbering, 175–179
AVG(), 60, 69

B

Bachman, Charles, 199
Barney, Mark E., 78
Bell Labs, 175
Ben-Nes, Michael, 89–90
BETWEEN predicates, 58, 66
  for hierarchical summaries, 69
  modifying, 59
Binary fractions, 123
Binary trees, 143
  balanced, 143–144
  complete, 143
  deletion from, 150
  finding ancestors in, 152
  finding descendants in, 152–153

Binary trees (*continued*)
finding parents on, 148
finding subtrees at nodes in, 149–150
insertion into, 150
multiway trees represented by, 154–155, 155f
perfect, 143
in procedural programming languages, 144
queries, 147–148
traversals, 145–147
Black, Paul E., 155
Borges, Jorge Luis, 193
Botelho, Flavio, 98–99

C

CASE expressions, 9, 20, 69
in disjoint hierarchies, 184
CAST( ) expression, 108
Catalan numbers, 143
Character-string data types, 5
CharIndex( ), 40
CHECK( ) constraint, 182
in disjoint hierarchies, 184
Child
encoding, 126–128
finding, 148
in recursive organization, 118–119
CinCom, 199
COBOL
CODASYL and, 200
IMS and, 203
v. SQL, 175, 213
Cobol
floating-point math in, 104
MVS, 14
recursion in, 13–14
CODASYL, 20
Codd, E. F., 17, 89
on auto-numbering, 177
COMPOUND function modifier, 174
Computed hybrid models, 161–164
CONNECT BY PRIOR, 169–170
Constraints
subqueries in, 37
uniqueness, 180–183
Control blocks, 202
Convergence point
breadth-first, 123, 127
depth-first, 123
COUNT statement, 50, 69
CREATE ASSERTION, 182
Cursors, 25–26
implementations, 114
Cycle, 4

D

Data
backing up, 201
integrity, 201
relationships, 208–209
requests, 204
Database description (DBD), 202
Databases. *See also* Relational databases
example, 204–205, 206–208
hierarchical, 199, 203
history of, 200–204
navigational, 199–200
network, 199
postrelational, 200
structure, 201
types of, 199–200
Datatypes, user-defined, 120
Date, Chris, 173
DDL constraints, 180–189
Decision trees, 197
Declarative Referential Integrity (DRI) actions, 186
DELETE FROM, 10, 62
Design considerations, 206
DevelopMentor, 43
Dewey Decimal Classification (DDC), 13, 192–193
of logic, 194t
Dewey, Melville Louis Kossuth, 192
Directed cycles, 167–168
Divisor parameter, 108
DL/I, 200, 202
calls, 203
control blocks, 200, 202
v. SQL, 213
Domain_key Normal Form (DNKF), 19
Double precision numbers, 103–104
DRI actions. *See* Declarative Referential Integrity actions

E

Edge enumeration model, 35, 42–43
Edges, 3–4
  dynamic, 13
  self-traversal, 7
  separation of, 92–94
  static, 12–13
ELSE clause, 62
Encoded node path, 123
  mapping, 126
Enumerated path
  advantages of, 128–129
  calculating, 128–132
Equality comparisons, 94
Equi-join, 177
EXPLODE operator, 173

F

FETCH statements, 213
Fields, 203, 211
  data, 212
  key, 211–212
  sequence, 211–212
FLOAT numbers, 103–104
Floating-point math, 104
  rounding errors in, 119
FLOOR( ) function, 152
Floyd-Warshall algorithm, 7
Forest, 4
FORTRAN, 5
Frammis, 69–70
Frequent insertion trees, 101
Function calls, 69
Function inverse, 130

G

Gaps, 102–103
  shifting, 114
Get Next within Parent (GNP) call, 210
GET-NEXT (GN) call, 210
GET-UNIQUE (GU) call, 210
Gilson, John, 152, 161–164
Graphs, 3
  acyclic, 120, 164
  adjacency arrays for, 6–7
  adjacency lists for, 6
  arrays for, 5
  connected, 4, 11
  convergent, 164–167
  cycles in, 36
  directed, 3
  general, 7–11, 164–168
  modeling, 4–5
  order of, 4
  theory, 3
  undirected, 3
Greatest common divisor (GCD) algorithms, 119
GROUP BY clauses
  for hierarchical summaries, 69
  of Tillquist and Kuo, 173–174

H

HAVING clause, 98
Heaps, 143, 150–153
Hierarchical data paths, 209–210
Hierarchical encoding schemes, 12, 191
  strengths and weaknesses of, 193
Hierarchical reports, 133
Hierarchical sequence, 209
Hierarchical summaries, 69
Hierarchies
  building, 132–133
  disjoint, 179, 183–186
  equality comparisons in, 94
  generalization, 179
  overlapping, 179
  properties of, 11–12
  in SQL DDL, 175
  types of, 12–13, 179
Huber, Heinz, 113
Hybrid models, 159–164

I

IBM
  IMS from, 200
  System R of, 17
IDENTITY, 176, 213
IDMS, 200
IMS, 199
  application programs, 203
  components, 200, 201f
  data language 1, 200
  hierarchical data paths in, 209–210
  hierarchical sequence in, 209
  hierarchies, 203
  history, 200–204
  v. SQL, 17
  strengths and weaknesses, 203–204

IMS (*continued*)
supported languages, 203
XPath and, 43
IMS Transaction Manager (IMS TM), 202–203
Indegree, 4
Inheritance, 12
of subordination, 21
INSERT INTO statement, 8–9
auto-numbering and, 176–177
dependencies of, 108
INSERT statement
anomalies, 20–21
trees and, 77
INTEGER numbers
arithmetic, 126
as encoded node path, 123
Integers, 103
Introduction to Algorithms (Cormen, Leiserson, & Rivest), 7
Irrational numbers, 119
Iterative parts update, 69–75
Izaguirre, Alejandro, 85, 89

J

Java library, 99
Johnson algorithm, 7

K

Kelsey, Morgan, 79
Keys, 180–181
NOT NULL UNIQUE constraint as, 187
Klempert, Arne, 61
KnowledgeSeeker, 197
Knuth, Donald E., 154
Knuth's Art of Programming (Knuth), 154
Kuo, Feng-Yang, 173–174

L

Landis, E. M., 144
Leaf nodes
in binary trees, 150
deleting, 30, 65
finding, 48
removing, 28
LEAVE command, 83
LEFT, 45
LEFT OUTER JOINs, 27, 55
LEVEL, 32
in Oracle, 169
in SBD, 171–172
Levels, 31–32
finding, 50
in linear version of nested sets model, 140
numbering, 32–33
of subordinates, 50–56
Library of Congress, 193
LIKE predicates, 37
Linear version of nested sets model
deletion in, 138–140
insertion in, 138–140
levels in, 140
paths in, 140
Linking column, 17
LISP, 5
Logarithm functions, 148
base two, 151–152
Logical relationships, 209
Lookup tables, 113–117

M

Machine language instructions, 5
Mackey, Aaron J., 93
Many-to-many relationships, 186–189, 208
Mark-up languages
parsing data in, 138
XML, 43
Matrix design, 183
MAX(), 60, 69
in reorganization, 118
in scalar subqueries, 55
Medinets, David, 99
Message boards, multithreaded, 79–81
Metadata models, 175
Microsoft Data Shaping Service for OLE DB, 174
Microsoft extensions, 174
Military occupational skills (MOS), 86
MIN(), 60, 69
MOS. *See* Military occupational skills
MoveSubtree, 81–83, 84–85
MS-SQL Server Newsgroup, 88
Multisets, 96
MySQL, 98–99

N

NDL language specification, 200
Negative numbers, 130
Nested circles, 47f

Nested intervals model, 119–135
tree, 121f, 122f
Nested set with depth model, 160
Nested sets, 45–47, 46f
insertions in, 101
in other languages, 98–99
tables, 48
Nested sets model
on Cartesian plane, 164
closing gaps in trees with, 66–69
converting, 89–90
deleting nodes with, 60–61
deleting single nodes with, 63–65, 63f, 64f
deleting subtrees with, 60–61, 61–63
finding paths with, 58
finding relative position with, 58–59
finding subordinates in, 56
functions in, 59–60
inserting trees in, 77–80
moving subtrees in, 80–83, 84–85
number line as, 137–138
in PHP, 61
pruning sets of nodes with, 65–66
ranges in, 195
spreading pairs in, 110
updating trees in, 77–80
VIEW for, 138
Newsgroups, Internet, 13
Node enumeration, 35
Nodes, 3–4. *See also* Leaf nodes
in adjacency list model, 28
attaching attributes to, 93–94
child, 120
comparing, 94–98
deleting, 39–40, 60–61
deleting single, 63–65, 63f, 64f
descendants of, 134–135
distance between, 128–132
duplicated, 86
dynamic, 13
finding subtrees at, 149–150
inserting, 40
multiple, 93–94
pruning sets of, 65–66
separation of, 92–94
shifting, 113–117
squeezing, 111
static, 12–13
visualizing, 129
Normalization, 19, 208
NOT NULL UNIQUE constraint, 180
as a key, 187
NULL statements, 22
in reorganization, 114–117
in subordinate adjacency lists, 158–159
using, 25
Number line
intervals on, 47f
as nested sets model, 137–138
NUMERIC(p,s) numbers, 104
NVARCHAR(n), 178–179

O

Object-oriented (OO) programming, 175
database, 179
extensions, 179
One-to-many relationships, 180, 186–189, 208
One-to-one relationships, 186–189
Online Computer Library Center, Inc. (OCLC), 192–193
OO programming. *See* Object-oriented programming
Oracle, 17
tree extensions, 169–171
ORDER BY, 58, 170
Organizational charts, 18f
characteristics of, 13
hierarchies and, 11–12
Outdegree, 4
Outer join query, 89–90, 98
OUTER JOINs, 178

P

Parents
encoding, 126–128
finding, 148
in recursive reorganization, 118–119
Partial order mappings, 120–23
Parts explosions
accumulated totals in, 60
hierarchies and, 11–12
in nested sets model, 53f
numbering, 46
Pascal, 5
IMS and, 203

Path encoding function, 130–132
Path enumeration models, 35
Path enumeration table, 165–167
Paths, 4
  in convergent graphs, 164–167
  finding, 50, 58
  in linear version of nested sets model, 140
  removing, 9–10
  splitting, 40–42
PHP
  MySQL and, 98–99
  nested sets model in, 61
PL/I, 5
  IMS and, 203
  v. SQL, 213
Pointer chains, 4–5
  binary trees and, 144
  faking, 89
  in TOTAL, 199–200
POSITION(), 40
Postgres newsgroups, 99
Preorder tree transversal algorithm, 47
PREVIOUS (column) function, 171–172
PRIMARY KEY, 181, 186–189
  uniqueness constraints and, 180–181
Procedural languages, 17, 25–26
  adjacency model and, 34
  binary trees in, 144
  influence of, 174
Program specification block (PSB), 202
Promotion
  horizontal, 60
  in nested sets model, 64, 64f
  vertical, 60
Pushdown stack algorithm
  in adjacency list model conversion, 90
  in nested sets model conversion, 89

Q

Queries
  based on subtrees, 52
  binary tree, 147–148
  optimizing, 48

R

Rational numbers, 119–135
READ() command, 213
REAL numbers, 103–104
Records, 203, 210–211
  segments of, 211
Recursion, 13–15
  reorganization with, 118–119
Recursive parts update, 76–77
Recursive structures, 15
REFERENCES clause, 189
  Declarative Referential Integrity, 20
  in shop categories, 196
Relational Database: Selected Writings (Date), 173
Relational databases, 199
  arguments against, 17
  MySQL and, 98
Relative position, 58–59
Reorganization
  with lookup table, 113–117
  partial, 109–111
  procedures, 108
  with recursion, 118–119
  total, 113–119
REPLACE() function, 36, 37, 167
RETURN statement, 124
REXX, 203
RIGHT, 45
Right interval boundary, 133
Romley, Richard, 54, 118
Rounding functions, 108
Rozenshtein, David, 31–32

S

Santayana, George, 213
Scott/Tiger database, 17–18
Searches
  breadth-first, 145, 146
  depth-first, 146
  paths, 210, 210f
Segment
  child, 212
  definition, 213–214
  dependent, 212
  format, 211–212
  parent, 212
  retrieval, 210
  root, 212
  twin, 212
Segment search arguments (SSAs), 204

SELECT statements
finding subordinate levels and, 51
in MySQL, 98
traversals and, 169
Self-JOINs, 26–28, 49
finding subordinates with, 56
Self-references, 157–158
Sequence, 27–28
Sequential processing, 98
SET clause, 33
Set-oriented languages, 45
Shop categories, 194–197
Siblings
finding, 126–128
inserting, 104, 105f
swapping, 88–89
SIGNAL command, 83
Skonnard, Aaron, 43
Smith-Barney, 54
Spread, 102
computing, 104–107, 107t
rightward growth, 111
varying, 108
SQL
cursor syntax of, 25
v. other languages, 213
performance, 27
temporal model in, 119
SQL FORUM, 31
SQL-92
constraint names in, 70
subqueries in, 37
SQL-99
Array in, 7
OO extensions in, 179
WITH operator and, 172
user-defined datatypes in, 120
SQL/XML, 138
Squeezing, 111–112
SSAs. *See* Segment search arguments
START WITH, 169
Statistical tools, 197
Stern-Brocot Numbers, 156
Strings
padding, 38
searches with, 35
Stroustrup, Bjarne, 175
Structures
comparing, 94–98
multiple, 92–93
Subordinates, 56–57
of deleted nodes, 28
finding levels of, 50–56
promoting, 30, 39–40
searching for, 37–38
Subqueries
in constraints, 37
scalar, 61, 83
Subtrees, 11
in binary trees, 150
comparing, 97–98
deleting, 28–29, 39, 60–61, 61–63, 138–39
duplicating, 85–87
finding, 49, 149–150
inserting, 40, 139–140
moving, 80–83, 84–85
in nested sets model, 60
promoting, 30–31
queries based on, 52
Summary functions, 69
Sybase/SQL Server, 176
System R, 17

T

Table lookups, 109
Tables
auxiliary, 27, 80
heap, 148, 151
join, 180, 186
junction, 180, 186
nested set, 48
overhead and, 101
of sequential integers, 150
temporary, 80
Tillquist, John, 173–174
TOTAL, 199–200
Traversals
binary tree, 145–147
with CONNECT BY PRIOR, 170
in deleting subtrees, 28–29
depth-first, 161, 209
EXPLODE operator and, 173
inorder, 146–147
navigation and, 25–28

Traversals (*continued*)
postorder, 145, 162
preorder, 145–146, 161
SELECT statements and, 169
Trees. *See also* Binary trees; Decision trees; Frequent insertion trees
AVL, 143–144
B+, 145
B–, 144, 145
on Cartesian plane, 164
closing gaps in, 66–69
depth of, 37
extensions, 169–174
Fibonacci, 144
flattening, 54–55, 66
forest and, 4
height of, 50
height-balanced, 144
identical, 97
inserting, 77–89
as intervals on number line, 47f
levels, 50
moving subtrees within, 80–83, 83–85
multiway, 154–155, 155f
navigation in, 25–26
as nested circles, 47f
as nested sets, 46, 46f
path enumeration model of, 120
properties of, 11, 49
rebuilding, 39–40
as recursive structures, 15
Red-Black, 144
stretching, 111
summary functions on, 69
unique paths in, 35
updating, 77–89
weight-balanced, 144
TRIGGER, 21, 183
Tropashko, Vadim, 119, 120
Truncation functions, 108

U

UNION ALL, 80, 172–173
UNION query, 106
UNIQUE constraint, 18, 23, 181–182
UPC code, 195
UPDATE statement
anomalies, 19–20
moving subtrees and, 83
in reorganization, 110
SET clause of, 33
trees and, 77
U.S. Postal Service, 192

V

VARCHAR(n) strings, 35, 178–179
Vertices, 3
VIEW statements
finding subordinate levels and, 51
flattening trees with, 66
for nested sets model, 138
WITH operator and, 172
of spread, 104
tree height and, 50
UNION ALL and, 80
Virtual Storage Access Method (VSAM), 202
Vujnovic, Damjan S., 146
on binary tree queries, 147–148

W

Walk, 4
Walsh, Michel, 88
WHERE clauses, 62
aggregates in, 55
WITH operator, 172–173

X

XBD Systems, 171–172
XML, 43
XPath, 43

Y

Yao, S. Bing, 171

Z

ZIP codes, 191–192

ABOUT THE AUTHOR

Joe Celko is a Professor and Vice President of Relational Database Management at Northface University in Salt Lake City. He is a noted consultant, lecturer, and one of the most read SQL authors in the world. He is well known for his 10 years of service on the ANSI SQL standards committee, his column in *Intelligent Enterprise* magazine (which won several Readers' Choice Awards), and the war stories he tells to provide real-world insights into SQL programming. His best-selling books include *Joe Celko's SQL for Smarties: Advanced SQL Programming, Second Edition; Joe Celko's SQL Puzzles* and *Answers;* and *Joe Celko's Data and Databases: Concepts in Practice,* which are all published by Morgan Kaufmann.